Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Métodos Polinomiais para Simplificação de Fórmulas Modais

André Belle Menezes

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof.a Dr.a Cláudia Nalon

Brasília
2016

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Métodos Polinomiais para Simplificação de Fórmulas Modais

André Belle Menezes

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Cláudia Nalon (Orientador)
CIC/UnB

Prof. Dr. Guilherme Novaes Ramos
CIC/UnB

Dr. Díbio Leandro Borges
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 08 de dezembro de 2016

Dedicatória

Dedico este trabalho a minha família, se não fosse por ela, não estaria aqui. Este trabalho e meu diploma são para vocês. Dedico também a minha professora, Cláudia Nalon, que além de uma educadora exemplar, se mostrou uma pessoa incrível.

Agradecimentos

Este trabalho é fruto de muito suor, dedicação e companheirismo. Agradeço a Deus por dar força e fazer com que tudo cooperasse para um bom trabalho. Agradeço a minha professora que sempre esteve presente para tirar dúvidas, ajudar em todos os momentos e, mesmo quando eu merecia uns puxões de orelha, sempre foi carinhosa e me ensinou a como realizar minhas tarefas com paciência; meu muito obrigado à professora Cláudia Nalon, espero que você alcance seus sonhos e seja muito feliz.

Tenho que separar um momento para falar dos meus amigos, companheiros de UnB, aqueles que estiveram comigo desde o início dessa longa caminhada e foram um dos motivos de eu manter a motivação para terminar esse curso. Enfrentamos juntos professores inconvenientes, provas difíceis, trabalhos árduos, viagens longas até a Universidade, toda a pressão e desgaste psicológico que encontramos na faculdade; vocês são incríveis, espero poder contar com todos no resto da minha vida, o semestre de Ciência de Computação do segundo semestre de 2010 ainda será lembrado nesse país, pois nós somos fora de série. Muito obrigado pela amizade de vocês.

Agradeço a minha família que deu todo o suporte necessário que precisei para entrar e continuar na faculdade, sempre se preocupando com que eu tivesse o melhor em tudo, esse trabalho é graças a vocês.

Resumo

A simplificação de fórmulas lógicas busca melhorar a eficiência do tratamento destas. Este projeto desenvolve um algoritmo baseado na eliminação de literal puro e propagação de constante para realizar simplificação. O trabalho é realizado em cima do provador KSP que já implementa os métodos de simplificação citados, porém acreditamos que esse novo algoritmo desenvolvido trará maior desempenho na execução. Como melhorias foram propostas uma lista de localização para as proposições das fórmulas e a realização de eliminação de literal puro enquanto isso for possível. A implementação é explicada detalhadamente neste trabalho. As comparações realizadas entre as versões dos provadores mostra que a nova versão é melhor para algumas fórmulas e o KSP para outras. Apesar disso, a nova implementação pode produzir fórmulas mais simplificadas e sua manipulação das estruturas de dados é mais eficiente, porém os tempos de execução não foram afetados de maneira substancial.

Palavras-chave: lógica modal, raciocínio automático, simplificação

Abstract

Simplification of logical formulae is a preprocessing procedure which seeks to reduce the size of the input formula in order to improve the overall efficiency of theorem provers. This project implements two procedures for simplification: one for pure literal elimination and other for constant propagation. The work is based on an existing prover, KSP, which already implements the mentioned simplification methods. However, our hypothesis was that our implementation would improve performance by changing the underlying data structures. One of the implementation improvements proposed was a localization list for the propositions of the formulae. Also, pure literal elimination is executed until a fixed-point is reached. We explain the implementation in detail. Comparison between the provers versions show that there is no time difference pattern, but the new implementation usually produces results in less time. The new implementation produces better formulae in some cases and makes good use of data structures, but execution times have not been substantially affected.

Keywords: modal logic, automated reasoning, simplification

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos dias de hoje, fazemos de tudo para ganhar tempo: vamos à padaria de carro, contratamos alguém para lavar o carro, almoçamos no restaurante para não ter que cozinhar; qualquer segundo que ganhamos é comemorado. Agora imagine aquele problema gigantesco que você terá que resolver amanhã. Ao invés de dividí-lo em partes pequenas, não seria bom também diminuir o tamanho total dele? Essas são duas propostas deste trabalho: ganhar tempo e diminuir um problema. Sendo mais específico, o nosso trabalho consiste em tentar diminuir o tempo de execução de um provador de teoremas e simplificar as fórmulas de entrada deste provador.

Os problemas que estaremos lidando são problemas da Lógica Modal K [1]. Quem estará resolvendo estes problemas será o Provador de Teoremas KSP, apresentado em [2], que verifica a satisfatibilidade de uma fórmula lógica dada e retorna diversas informações para o usuário como número de cláusulas, número de símbolos proposicionais, grau modal da fórmula, tamanho e outras, que são úteis para direcionar o uso de técnicas de pré-processamento e prova.

O problema de satisfatibilidade da lógica modal K é PSPACE-completo [3], um nível de complexidade muito alto. Na computação, problemas em P são facilmente tratáveis, mas aqueles em NP são conhecidos pela sua dificuldade [4]. O conjunto de problemas PSPACE engloba-os, como mostra a Figura 1.1 (adaptada de [4]). Assim, o pré-processamento eficiente da entrada é desejável, pois há trabalhos que mostram uma diminuição do tempo de execução utilizando-se pré-processamento [5, 6]. O pré-processamento consiste em métodos que buscam diminuir o tamanho da fórmula antes do processamento desta.

Neste trabalho foi realizada a reimplementação dos métodos de eliminação de literal puro e de propagação de constantes para o provador citado. As novas implementações têm como objetivo manter o tempo polinomial dos métodos, manter a satisfatibilidade das fórmulas, evitar diversas passagens na árvore sintática, que representa as fórmulas de entrada, realizar mais simplificações do que o método de eliminação de literal puro atual

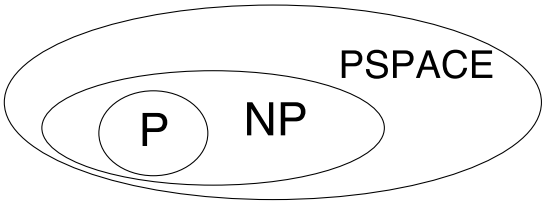


Figura 1.1: Classes de complexidade.

do KSP.

Para alcançar os objetivos, a metodologia utilizada foi estudar a Lógica Modal K, sua semântica e sintaxe, estudar a eliminação de literal puro e a propagação de constantes, além de conhecer o provador, suas estruturas e seu funcionamento. Após a base teórica estar consolidada, foi dado início à implementação dos métodos e realizados testes diversos para medir a eficiência alcançada e compará-la com o KSP para o levantamento de dados. Feito isso, os dados foram analisados, mostrando que a nossa implementação é tão ou um pouco mais eficiente do que a implementação anterior em grande parte dos casos.

O Capítulo 2 expõe a teoria necessária da Lógica Modal K para entendimento do trabalho. O Capítulo 3 apresenta as simplificações que foram implementadas no algoritmo que foi desenvolvido e o método de eliminação de literal puro. A implementação é descrita com detalhes no Capítulo 4. Em seguida, apresentamos os diversos experimentos e dados obtidos, bem como sua análise, no Capítulo 5. Por fim, o Capítulo 6 traz as conclusões e trabalhos futuros.

Capítulo 2

Base Teórica

A lógica em Ciência da Computação é utilizada como linguagem a fim de modelar situações do cotidiano de um profissional da Computação, para que seja possível analisá-las formalmente [7]. Analisar situações significa criar argumentos sobre elas; formalmente, os argumentos devem ser válidos e justificados ou executados em uma máquina [7]. Para exemplificar o uso da lógica, ela é atualmente utilizada na área de robótica para programação da Inteligência Artificial das máquinas e outras aplicações afins [8].

Os argumentos podem ser expressos através de linguagens naturais ou de linguagens formais. Uma linguagem natural é aquela utilizada para comunicação no cotidiano, como o português, inglês e linguagem de sinais; um problema dessas linguagens é que elas podem apresentar ambiguidade em suas sentenças, o que impede uma análise precisa destas [7]. Já uma linguagem formal é aquela que utiliza símbolos precisos e operacionais para evitar a ambiguidade, buscando ser concisa e exata [7].

Um argumento é um conjunto de sentenças, chamadas de premissas, seguidas de uma única sentença, a conclusão. Considere o seguinte argumento em português, uma linguagem natural:

Se chove e Maria não está com seu guarda-chuva, então ela se molha. Maria não está molhada. Está chovendo. Portanto, Maria está com seu guarda-chuva.

O argumento pode ser traduzido para a linguagem formal da seguinte maneira:

$$\begin{array}{c} p \ \wedge \neg q \rightarrow r \\ \neg r \\ p \\ \hline q \end{array}$$

onde p = *chove*, q = *Maria estar com o guarda-chuva* e r = *Maria se molhar*. Esse argumento é válido, pois sua conclusão (parte debaixo do argumento) está de acordo com as

premissas dadas (parte de cima do argumento). Abstraíram-se os fatos do acontecimento da chuva e de Maria estar ou não com seu guarda-chuva com o uso de símbolos. Essa é a estrutura lógica desse argumento. A estrutura lógica é de vital importância para o estudo da lógica, pois organiza o argumento em partes bem definidas que podem ser exploradas sem ambiguidade [7].

2.1 Lógica Modal

Agora imagine que Maria está na Terra e Armstrong é um astronauta na superfície de Marte. Suponha que é possível chover em Marte e sempre que chove, a chuva ocorre na Terra e em Marte. Então, se está chovendo, Armstrong não se molha, pois tem sua roupa de astronauta; Maria, porém, necessita de um guarda-chuva para não se molhar. Como está chovendo na Terra, então também chove em Marte. Maria e Armstrong não estão molhados, então Maria está com um guarda-chuva. Temos essa situação ilustrada na Figura 2.1.

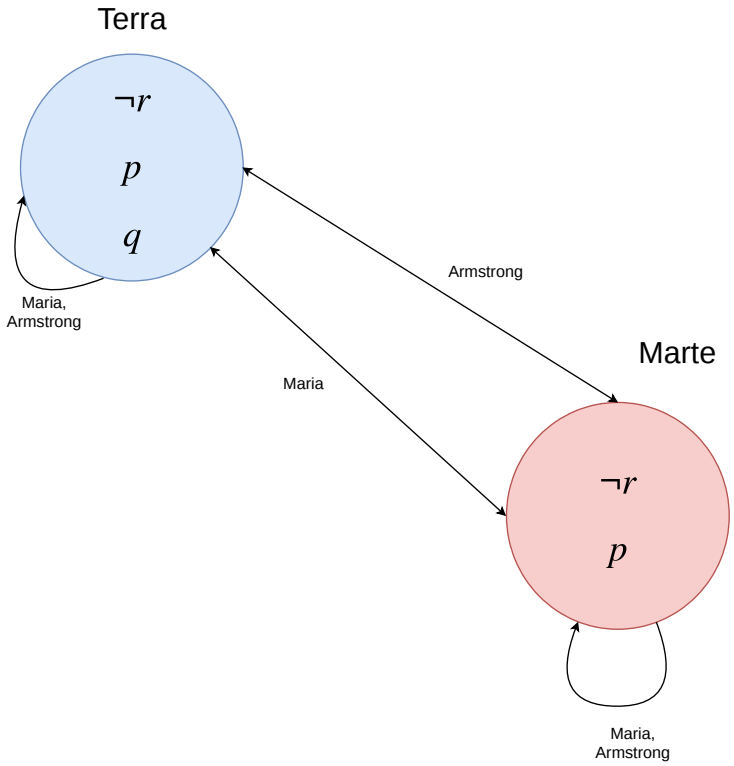


Figura 2.1: Exemplo com a Terra e Marte

Com a entrada de um novo mundo no nosso exemplo, pode-se criar novas relações entre os mundos e os agentes presentes: Maria e Armstrong. Ou seja, temos um universo - conjunto de mundos e agentes [1]. No exemplo da Figura 2.1, vê-se que p (chuva) e $\neg r$ (não estar molhado) são comuns entre os mundos, já q (estar com um guarda-chuva) ocorre apenas na Terra e para Maria.

A linguagem formal que permite essa análise envolvendo raciocínio sobre mundos e agentes é a Lógica Modal, que será a base para o desenvolvimento deste trabalho. Nas próximas seções, essa lógica será apresentada formalmente.

2.1.1 Sintaxe

A sintaxe de uma linguagem formal é o conjunto de regras e símbolos que serão utilizados para formar sentenças da linguagem [9].

A Lógica Modal utiliza os símbolos lógicos da Lógica Proposicional clássica para expressar seus argumentos e também acrescenta novos operadores. São eles [7]:

1. $\mathcal{P} = \{p, q, r, ...\}$, um conjunto enumerável de símbolos;
2. $\{ \neg, \vee, \wedge, \rightarrow, \leftrightarrow \}$, um conjunto de operadores clássicos;
3. $\{ (,) \}$, símbolos de pontuação;
4. $\{\Box_i, \Diamond_i\}$, um conjunto de operadores modais, para $i \in A = \{1, 2, ..., n\}$ onde cada i é um agente observador;
5. $\bot$ e $\top$, constantes lógicas.

Os elementos do conjunto $\mathcal{P}$ são chamados de símbolos proposicionais. Os elementos do Item 2 são os conectivos lógicos clássicos. O Item 5 refere-se às contantes lógicas, onde $\bot$ representa a constante de valor Falso e $\top$ representa a constante de valor Verdade. Os novos operadores são aqueles apresentados no Item 4. O operador $\Box_i$ é usualmente utilizado para denotar "necessidade" e o seu dual, $\Diamond_i$, para denotar "possibilidade".

A lógica modal possui vários sistemas, sendo que a diferença entre eles está na imposição ou não de restrições sobre o conjunto de relações entre mundos [1]. Este trabalho será desenvolvido com a versão multimodal da lógica básica K, cuja linguagem é definida a seguir [6]:

Definição 1 *O conjunto de Fórmulas Bem Formadas (FBF) de* K *é dado por:*

1. Se $p \in \mathcal{P}$*, então* $p \in FBF$*;*

2. Se $\phi \in FBF$*, então* $\neg\phi \in FBF$*;*

3. Se $\phi \in FBF$ *e* $\psi \in FBF$*, então também estão* $\neg\phi, (\phi \vee \psi), (\phi \wedge \psi), (\phi \rightarrow \psi), (\phi \leftrightarrow \psi), (\Box_i\phi)$ *e* $(\Diamond_i\phi)$ *para* $i \in A = \{1, 2, ..., n\}$*.*

Definição 2 *Uma fórmula atômica é uma fórmula que não possui conectivos lógicos da linguagem e que, consequentemente, não possui subfórmulas [9].*

Por exemplo, p e q são fórmulas atômicas. A fórmula $\Box_1(p \rightarrow q) \vee \neg p$ é um exemplo de fórmula bem-formada e que não é atômica.

Esses conceitos são necessários para a construção de fórmulas na Lógica Modal. A seguir, será mostrada a semântica dessa lógica, ou seja, como cada fórmula é avaliada.

2.1.2 Semântica

Para iniciar a semântica, continuaremos a utilizar a Figura 2.1, a qual será analisada de acordo com os conceitos semânticos da Lógica Modal. O que foi tratado como universo naquele exemplo, chamaremos de *Estrutura de Kripke* ou *Modelo de Kripke* (adaptado de [6]).

Definição 3 *Uma Estrutura de Krikpe μ para n agentes é uma tupla $\langle S, s_0, \pi, R_1, \ldots, R_n \rangle$ onde S é o conjunto de mundos possíveis e $s_0 \in S$; a função de avaliação $\pi : S \longrightarrow (\mathcal{P} \longrightarrow \mathcal{V})$, com $\mathcal{V} = \{verdadeiro,\ falso\}$, $s \in S$, indica que cada proposição em cada mundo possui seu valor verdade; e R_i é uma relação binária em S ($R_i \subseteq S \times S$), para $i = 1, 2, \ldots, n$.*

A Estrutura de Kripke correspondente ao exemplo apresentado na Figura 2.1 é:

$$\mu = \langle \{Terra, Marte\}, Terra, \pi, R_{Maria}, R_{Armstrong} \rangle$$

onde Maria e Armstrong são os agentes dessa estrutura[1]. A função de avaliação é definida como $\pi(Terra)(p) = \pi(Marte)(p) = verdadeiro$, $\pi(Terra)(q) = verdadeiro$, $\pi(Marte)(q) = falso$ e $\pi(Terra)(r) = \pi(Marte)(r) = falso$; e as relações entre mundos é dada por $R_{Maria} = R_{Armstrong} = \{(Terra, Marte), (Marte, Terra), (Terra, Terra), (Marte, Marte)\}$.

Veja que a função de avaliação atribui *verdadeiro* a p na Terra, sendo que p corresponde ao fato de chover. Formalmente, dizemos que p é *satisfeito* na Terra no modelo μ. Em símbolos, a relação de satisfação é denotada por $\models$. Para este exemplo específico, quando queremos dizer que *chove na Terra*, escrevemos:

$$(\mu, Terra) \models p$$

[1]Tomamos a liberdade de nomear os agentes como no exemplo corrente, ao invés de utilizar os números naturais como índices das relações.

Note que a relação de acessibilidade $R_{Armstrong}$ nos diz que o agente Armstrong pode ver todos os mundos. Quando Armstrong está na Terra, ele consegue ver que o que acontece em Marte e na Terra, ou seja, que p também é satisfeito nestes mundos. Para Armstrong, p é *necessário* neste modelo μ, pois é verdade em todos os mundos que lhe são acessíveis. O operador $\Box_{Armstrong}$ é usado para denotar necessidade. Para o exemplo corrente temos:

$$(\mu, Terra) \models \Box_{Armstrong}\, p$$

Ao contrário de p, q é verdade apenas na Terra. Para a agente observadora Maria, q acontece na Terra e, em Marte, q não é verdade. Então q é *possível* na Terra em μ. Utilizando os símbolos modais, temos:

$$(\mu, Terra) \models \Diamond_{Maria}\, q$$

Para Maria, q é possível em μ, pois em algum mundo acessível por Maria q é verdade e em algum outro mundo que Maria tem acesso, q é falso.

2.1.3 Satisfatibilidade

Para analisar semanticamente as fórmulas, necessitamos entender o significado de satisfatibilidade. Para a lógica modal, deve-se analisar a Estrutura de Kripke em que a fórmula está inserida, os agentes e mundos envolvidos. Apresentamos agora a definição formal, adaptadas de [1].

Definição 4 *Uma fórmula ϕ é satisfeita no mundo α de uma estrutura de Kripke μ se:*

- $(\mu, \alpha) \models \phi$*, se, e somente se,* $\pi(\alpha) : \phi = verdadeiro$*;*
- $(\mu, \alpha) \models \neg\phi$*, se, e somente se,* $(\mu, \alpha) \not\models \phi$ *;*
- $(\mu, \alpha) \models \phi \ \wedge \ \psi$*, se, e somente se,* $(\mu, \alpha) \models \phi$ *e* $(\mu, \alpha) \models \psi$*;*
- $(\mu, \alpha) \models \phi \ \vee \ \psi$*, se, e somente se,* $(\mu, \alpha) \models \phi$ *ou* $(\mu, \alpha) \models \psi$*;*
- $(\mu, \alpha) \models \phi \rightarrow \psi$*, se, e somente se, se* $(\mu, \alpha) \models \phi$ *então* $(\mu, \alpha) \models \psi$*;*
- $(\mu, \alpha) \models \phi \leftrightarrow \psi$*, se, e somente se,* $(\mu, \alpha) \models \phi$ *se, e somente se,* $(\mu, \alpha) \models \psi$*;*
- $(\mu, \alpha) \models \Box_i\phi$*, se, e somente se, para todo mundo* β *tal que* $\alpha R_i \beta$*, então* $(\mu, \beta) \models \phi$*;*
- $(\mu, \alpha) \models \Diamond_i\phi$*, se, e somente se, para algum mundo* β *tal que* $\alpha R_i \beta$*, então* $(\mu, \beta) \models \phi$*.*

Dizemos que uma fórmula é *satisfatível* se existe uma Estrutura de Kripke μ tal que $(\mu, s_0) \models \phi$.

2.1.4 Validade

O conceito de validade possui semelhança com o de satisfatibilidade, porém estes possuem uma sutil diferença. Uma fórmula é satisfatível se é satisfeita em algum modelo. Uma fórmula é válida se é satisfeita em todos os modelos. Formalmente:

Definição 5 *Seja ϕ uma fórmula bem formada. ϕ é válida se, e somente se, para qualquer Estrutura de Kripke μ, temos $(\mu, s_0) \models \phi$.*

Toda fórmula válida é satisfatível, mas nem toda fórmula satisfatível é válida.

2.1.5 Equivalência

Para realizar simplificação em uma fórmula, o conceito de equivalência é essencial, pois se uma fórmula ϕ é simplificada e gera uma fórmula ψ, então queremos que os modelos que satisfazem ϕ também satisfaçam ψ.

Definição 6 *Duas fórmulas ϕ e ψ são equivalentes se, e somente se, para toda Estrutura de Kripke μ temos $(\mu, s_0) \models \phi$ se, e somente se, $(\mu, s_0) \models \psi$.*

Capítulo 3

Simplificação

A simplificação busca reduzir o tamanho de uma fórmula sem alterar sua satisfatibilidade [2]. A utilização da simplificação de fórmulas pode trazer melhoras significativas em relação ao tempo gasto para decidir a satisfatibilidade ou não de uma fórmula [6]. Nesta seção, serão mostradas algumas formas de simplificação e estudos mostrando a efetiva melhora com esse método, seja diminuindo o número de cláusulas ou apenas diminuindo o tamanho destas.

3.1 Simplificações

As simplificações apresentadas nesta seção são bem intuitivas e simples, porém são valiosas pela diminuição de literais e cláusulas da fórmula. As regras de reescrita que simplificam as fórmulas são as seguintes (adaptado de [9]):

1. Substituir toda subfórmula na forma $(\psi \vee \psi)$ por ψ;
2. Substituir toda subfórmula na forma $(\psi \wedge \psi)$ por ψ;
3. Substituir toda subfórmula na forma $(\psi \vee \neg\psi)$ por $\top$;
4. Substituir toda subfórmula na forma $(\psi \wedge \neg\psi)$ por $\bot$;
5. Substituir toda subfórmula na forma $(\psi \vee \top)$ por $\top$;
6. Substituir toda subfórmula na forma $(\psi \wedge \bot)$ por $\bot$;
7. Substituir toda subfórmula na forma $(\psi \vee \bot)$ por ψ;
8. Substituir toda subfórmula na forma $(\psi \wedge \top)$ por ψ;
9. Substituir toda subfórmula na forma $(\Box_i \top)$ por $\top$;
10. Substituir toda subfórmula na forma $(\Diamond_i \bot)$ por $\bot$;

11. Substituir toda subfórmula na forma $(\neg\bot)$ por $\top$;
12. Substituir toda subfórmula na forma $(\neg\top)$ por $\bot$;
13. Substituir toda subfórmula na forma $(\top \to \psi)$ por ψ;
14. Substituir toda subfórmula na forma $(\bot \to \psi)$ por $\top$;
15. Substituir toda subfórmula na forma $(\psi \to \top)$ por $\top$;
16. Substituir toda subfórmula na forma $(\psi \to \bot)$ por $\neg\psi$;
17. Substituir toda subfórmula na forma $(\psi \leftrightarrow \bot)$ por $\neg\psi$;
18. Substituir toda subfórmula na forma $(\psi \leftrightarrow \top)$ por ψ.

3.2 Eliminação de Literal Puro

Um tipo de simplificação que preserva satisfatibilidade é a eliminação de literal puro [5]. Ela consiste em substituir literais por constantes lógicas e assim aplicar simplificações à fórmula resultante.

Para definir literal puro, precisamos do conceito de polaridade, apresentado a seguir.

3.2.1 Polaridade

A polaridade de uma fórmula é determinada a partir do número de negações sob a qual esta fórmula se encontra [5]. Se estiver sob um número par de negações, sua polaridade é positiva; se estiver sob um número ímpar de negações, sua polaridade é negativa. Se a mesma subfórmula ocorrer tanto positiva quanto negativamente, então sua polaridade é definida como zero. Segue a definição formal (adaptado de [5]).

Definição 7 *Sejam ϕ, ψ, γ fórmulas bem formadas. A polaridade é definida por casos:*

- $\mathsf{pol}(\phi) = 1$*;*
- *se* $\phi = p$*, então* $\mathsf{pol}(p) = \mathsf{pol}(\phi)$*;*
- *se* $\phi = \neg\psi$*, então* $\mathsf{pol}(\psi) = -\mathsf{pol}(\phi)$*;*
- *se* $\phi = \psi \vee \gamma$*, então* $\mathsf{pol}(\psi) = \mathsf{pol}(\phi)$ *e* $\mathsf{pol}(\gamma) = \mathsf{pol}(\phi)$*;*
- *se* $\phi = \psi \wedge \gamma$*, então* $\mathsf{pol}(\psi) = \mathsf{pol}(\phi)$ *e* $\mathsf{pol}(\gamma) = \mathsf{pol}(\phi)$*;*
- *se* $\phi = \psi \to \gamma$*, então* $\mathsf{pol}(\psi) = -\mathsf{pol}(\phi)$ *e* $\mathsf{pol}(\gamma) = \mathsf{pol}(\phi)$*;*
- *se* $\phi = \psi \leftrightarrow \gamma$*, então* $\mathsf{pol}(\psi) = 0$ *e* $\mathsf{pol}(\gamma) = 0$*;*

- *se* $\phi = \Box_i\psi$, *então* $\mathsf{pol}(\psi) = \mathsf{pol}(\phi)$;
- *se* $\phi = \Diamond_i\psi$, *então* $\mathsf{pol}(\psi) = \mathsf{pol}(\phi)$;

Aplicando a definição acima a uma fórmula podemos calcular a polaridade de todas as suas subfórmulas. Por exemplo, suponha que $\phi = \Box_i p \rightarrow q \wedge \neg p$. Porque ϕ não está sob nenhuma negação, então sua polaridade é positiva. Ou seja, $\mathsf{pol}(\phi) = 1$. Para facilitar a apresentação do exemplo, usaremos índices distintos para o símbolo proposicional p que ocorre duas vezes. Aplicando a Definição 7 a ϕ, temos que:

1. $\mathsf{pol}(\Box_i p_1) = -\mathsf{pol}(\phi) = -1$ e $\mathsf{pol}(q \wedge \neg p_2) = \mathsf{pol}(\phi) = 1$;
2. aplicando mais uma vez a definição às fórmulas obtidas no item anterior, temos que $\mathsf{pol}(p_1) = -1$, $\mathsf{pol}(q) = 1$ e $\mathsf{pol}(\neg p_2) = 1$;
3. por fim, temos que $\mathsf{pol}(p_2) = -1$.

Para essa fórmula ϕ, p tem duas ocorrências e ambas possuem polaridade negativa; já q possui uma ocorrência e esta possui polaridade positiva.

Um literal é um símbolo proposicional ou sua negação [9]. Um literal é puro quando todas as aparições deste literal em uma fórmula possuem uma única polaridade. Suponha que p é um literal e que p ocorra em ϕ, onde ϕ é uma fórmula bem formada. Conforme [10], um literal p é puro se para toda ocorrência de p em ϕ, $pol(p) = 1$ ou se $pol(p) = -1$.

Exemplo *Na fórmula*

$$\phi = p \vee q \wedge \Box\neg q$$

o literal p é puro, pois sua única aparição ϕ tem polaridade positiva. Já q possui duas aparições e cada uma delas possui uma polaridade diferente: a primeira é positiva e a segunda é negativa.

A substituição de um literal p por q em uma fórmula ϕ consiste em trocar todas as ocorrências de p em ϕ por q. A eliminação de literal puro diz que se um literal p ocorre em uma fórmula ϕ apenas com uma polaridade, este literal pode ser substituído por uma constante lógica. A constante lógica será $\top$, caso p possua apenas polaridade positiva; ou por $\bot$, caso p possua apenas polaridade negativa. Esta substituição não altera a satisfatibilidade da fórmula ϕ [10].

Continuando com o exemplo acima, como o literal p é puro, podemos aplicar a eliminação de literal puro, substituindo p por $\top$, da seguinte forma:

$$\begin{aligned} & p \vee q \wedge \Box_i\neg q \\ \Rightarrow \quad & \top \vee q \wedge \Box_i\neg q \end{aligned}$$

Veja que a fórmula resultante permite a aplicação de simplificação, pois a constante lógica $\top$ aparece em um dos lados de uma disjunção. Assim, a disjunção é simplificada para a constante lógica, conforme visto na Seção 3.1, Item 5:

$$\begin{aligned} & \top \vee q \wedge \Box_i \neg q \\ \Rightarrow \quad & \top \end{aligned}$$

Capítulo 4

Implementação

Este capítulo explica como foi desenvolvida a implementação das simplificações estudadas no capítulo anterior. O trabalho foi desenvolvido alterando o provador de teoremas já existente, KSP [2]. Este provador já implementa simplificação e eliminação de literal puro, porém a implementação proposta nesse trabalho provê uma melhora de desempenho, o que será discutido no próximo capítulo.

A implementação realizada buscou uma maior eficiência na manipulação dos dados na tentativa de diminuição do tempo total para o processamento da saída do KSP, adicionando uma lista [11] de localização para as proposições, buscando evitar percorrimentos desnecessários na árvore sintática que representa a fórmula. O código-fonte deste projeto pode ser encontrado no endereço `http://www.cic.unb.br/~nalon/software/provadorAndre_2017.01.31.tar.gz`. Iremos aprofundar no funcionamento do provador adiante.

4.1 Provador KSP

O KSP é um programa que verifica a satisfatibilidade de uma fórmula lógica dada e retorna diversas informações para o usuário como número de cláusulas, número de símbolos proposicionais, grau modal da fórmula, tamanho e outras, que são úteis para direcionar o uso de técnicas de pré-processamento e prova. Para o processamento da entrada fornecida pelo usuário, o provador realiza vários procedimentos como transformar a fórmula na Forma Normal Negada e realizar diversos tipos de simplificação. O usuário pode escolher os procedimentos que deseja que sejam executados, bastando indicar a diretiva correspondente para este procedimento. A seleção de quais métodos o usuário quer utilizar é realizada através da linha de comando ou por meio de um arquivo de configuração. Para entrar com um arquivo de configuração, basta usar a diretiva *-c* e em seguida o arquivo de configuração desejado.

Os arquivos de configuração utilizados para o levantamento de dados deste projeto variam de acordo com o teste realizado e serão mostrados no momento da descrição dos testes, no Capítulo 5.

O KSP e as implementações realizadas neste trabalho são desenvolvidos na linguagem C. Mais informações podem ser encontradas em [2].

4.2 Árvores e Registros

O KSP utiliza a estrutura de dados de árvore [11], pois esta estrutura possui o conceito de hierarquia, utilizado para tratar as fórmulas lógicas, pois em uma fórmula como, por exemplo, $p \rightarrow q$, as subfórmulas p e q são filhas de $\rightarrow$.

O elemento básico de uma árvore é um nó, onde cada nó guarda uma informação e pode estar ligado a outros nós de forma hierárquica, ou seja, nós pais e nós filhos. No nosso caso, cada nó será uma proposição ou símbolo lógico.

Considere a seguinte fórmula

$$\Box_i(p \wedge \Diamond_i q) \vee \top \vee r \rightarrow \top$$

O KSP percorre cada símbolo da fórmula e constrói um nó para guardá-lo. A árvore construída para essa fórmula seria correspondente à dada na Figura 4.1.

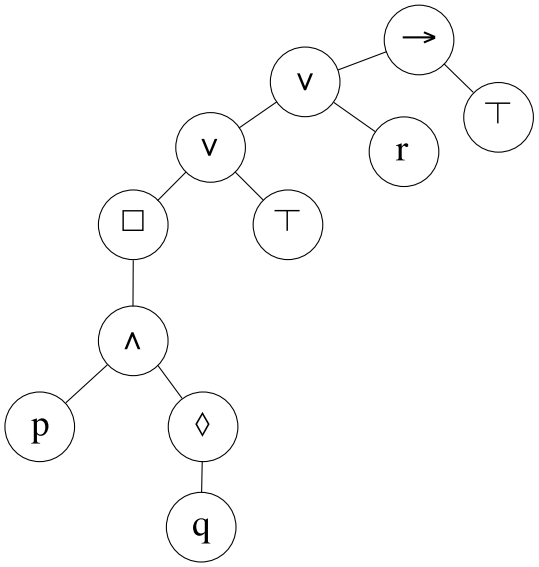
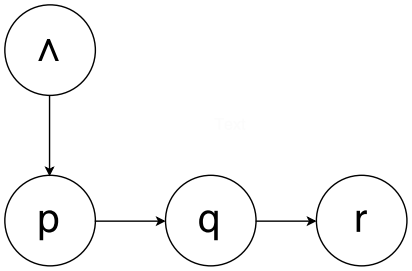


Figura 4.1: Árvore para a fórmula $\Box_i(p \wedge \Diamond_i q) \vee \top \vee r \rightarrow \top$

Note que a hierarquia dos nós para as fórmulas modais que estamos trabalhando seguirão sempre o mesmo padrão de acordo com o tipo da subfórmula, por exemplo, uma implicação ($\rightarrow$) sempre terá dois nós filhos: o filho da esquerda e o da direita. Os casos onde um nó terá mais de dois filhos são a conjunção ($\wedge$) e a disjunção ($\vee$). Estes operadores são tratados de uma forma particular e terão uma lista de nós filhos, que será detalhada posteriormente. As possibilidades para cada fórmula bem formada de K são:

1. $\wedge$ e $\vee$: uma lista de nós filhos. Exemplo na Figura 4.2.



-

Figura 4.2: Conjunção $p \wedge q \wedge r$

2. $\rightarrow$ e $\leftrightarrow$: dois filhos, um nó representando o que está à direita e outro representando o que está à esquerda. Exemplo na Figura 4.3.

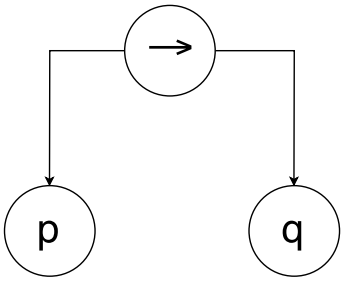


Figura 4.3: Implicação $p \rightarrow q$

3. $\neg, \Box_i, \Diamond_i$: um nó filho, considerado o filho da esquerda. Exemplo na Figura 4.4.

Agora que entendemos a organização da árvore, vamos aprofundar na estrutura da árvore. O nó da árvore possui um tipo, esse tipo é um *registro* que possui todos os

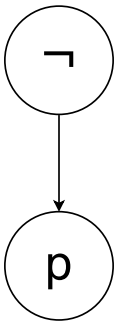


Figura 4.4: Negação $\neg p$

atributos que são importantes e necessitam ser guardados. Um registro é um conjunto de variáveis que pode ser heterogêneo, onde cada variável é um campo do registro [11].

A implementação da árvore no KSP possui dois tipos de registros: tnode e formulalist. O tnode representa os símbolos lógicos e as proposições da fórmula; o formulalist é um tipo criado especialmente para guardar as listas de filhos da conjunção e da disjunção.

Os campos de tnode, representado na Figura 4.5, são:

- int type: guarda o tipo da subfórmula;
- int id: cada tipo possui um identificador que o caracteriza;
- tnode *left: ponteiro para o nó filho da esquerda;
- tnode *right: ponteiro para o nó filho da direita;
- tnode *up: ponteiro para o nó pai, implementado neste trabalho;
- formulalist *list: ponteiro para a lista de nós filhos (tnode).

Os campos do registro formulalist, representado na Figura 4.6 são:

- tnode *formula: aponta para a estrutura tnode que guarda as informações do nó;
- formulalist *next: aponta para o próximo nó da lista.

Como visto, o formulalist funciona como uma estrutura de dados de lista simplesmente encadeada [11], pois aponta apenas para o próximo item da lista. O conteúdo que formulalist guarda é o tnode, que representa uma subfórmula. A Figura 4.7, que representa a fórmula $\bot \vee \neg p \rightarrow \top$, irá mostrar a estrutura da árvore utilizando as estruturas de dados aqui definidas para facilitar o entendimento. O campo `type` armazena constantes

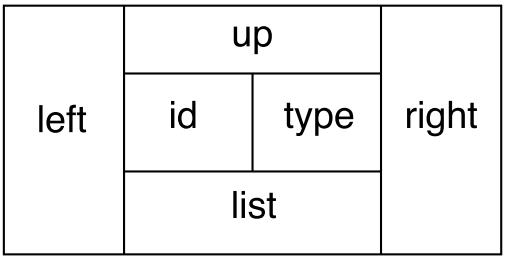


Figura 4.5: Registro tnode

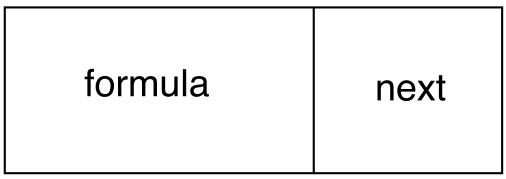


Figura 4.6: Registro formulalist

definidas na implementação do KSP, onde cada constante representa um símbolo lógico. O campo `id` guarda ou o tipo da fórmula (no caso de operadores clássicos), número do agente (no caso de operadores modais) ou o número inteiro associado com cada símbolo proposicional. A Tabela 4.1 mostra o que cada número indica no provador.

O campo `id` para proposições é específico: cada proposição possui um número identificador diferente. Seja $p \wedge q$ a entrada do KSP, todas as aparições de p possuirão o mesmo `id`; o mesmo para q, porém seu `id` será diferente do `id` de p. As constantes lógicas podem ter três valores de `id`: -1 indica constante de valor $falso$ ($\bot$), 0 é a constante de início,

Type	Id	Símbolo
0	id	Proposição
1	id	Box ($\Box$)
2	id	Diamante ($\Diamond$)
3	3	Negação ($\neg$)
4	4	Conjunção ($\wedge$)
5	5	Disjunção ($\vee$)
6	6	Implicação ($\rightarrow$)
7	7	Dupla implicação ($\leftrightarrow$)
8	-1, 0 ou 1	Constantes lógicas

Tabela 4.1: Tabela de definição de tipos para o KSP

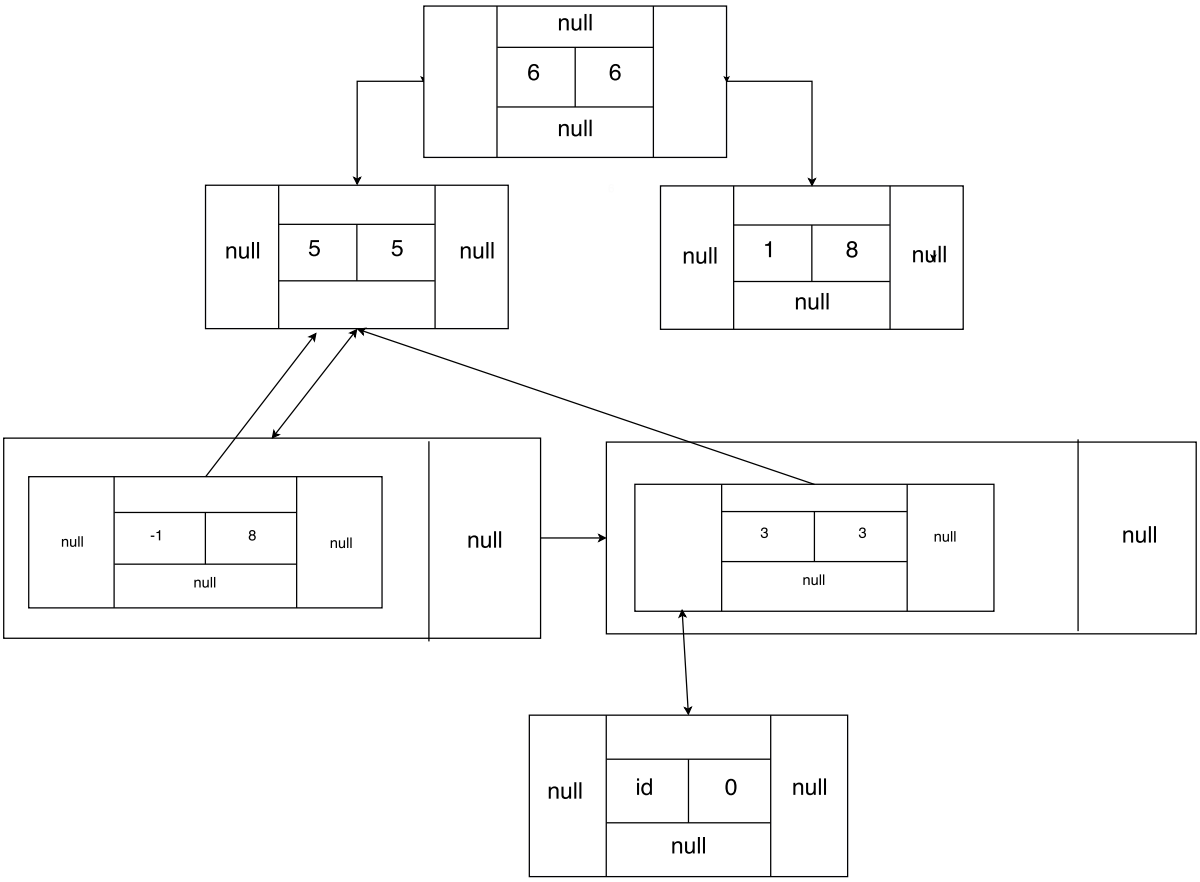


Figura 4.7: Árvore da fórmula $\bot \vee \neg p \rightarrow \top$ com registros

utilizada para transformar uma fórmula na forma normal em outras etapas do provador, e 1 indica constante de valor *verdadeiro* ($\top$). O campo `id` para os operadores modais guarda o índice correspondente ao agente.

4.3 Construção da Árvore

O KSP implementa sua própria gramática. Há um analisador léxico e um analisador sintático para tratar a fórmula de entrada. A construção da árvore é feita durante análise sintática, que é realizada de forma ascendente. Em cada regra da linguagem que possui um símbolo da fórmula, o respectivo nó da árvore é criado. Uma melhoria feita neste trabalho foi a inclusão de nós pais no registro que representa o nó da árvore. Suponha que a fórmula de entrada seja $\Diamond_i p$; em algum momento o analisador sintático chegará a regra da gramática *proposition*, a regra da gramática que trata proposições, e neste

momento criará o nó tnode p, que também será inserido na tabela de símbolos (explicada mais à frente). A próxima etapa é chegar na regra *DIAMOND proposition*, neste momento, o nó tnode $\Diamond_i$ será criado e o pai da *proposition* p será configurado como sendo o nó que contém o operador modal, $\Diamond_i$.

As outras fórmulas bem formadas são tratadas similarmente. A maior diferença é nas regras de conjunção e disjunção, onde, além de criar os tnode's necessários, há a criação das suas listas de filhos com a estrutura formulalist.

4.4 Tabela de Símbolos

Uma outra estrutura muito importante para o desenvolvimento deste projeto é a Tabela de Símbolos. Assim como a construção da árvore é feita na etapa de análise sintática, a construção da Tabela de Símbolos também. Existe uma regra na sintaxe da linguage do KSP que define proposição (conforme exemplo acima). Quando o analisador sintático faz o emparelhamento com esta regra, cria-se um nó para guardar as informações da proposição na Tabela de Símbolos. O tipo do nó de uma proposição é um registro nomeado prop_node. Os campos do registro prop_node utilizados nesse trabalho são:

- char *name: guarda o nome da proposição;
- int id: id para identificar cada proposição;
- int occur_positive: contador para identificar o número de ocorrências da proposição com polaridade positiva;
- int occur_negative: contador para identificar o número de ocorrências da proposição com polaridade negativa;
- location_list *locationlist: guarda as posições da proposição na árvore da fórmula.

O campo *locationlist não existia na implementação original do KSP. A Figura 4.8 representa o registro prop_node.

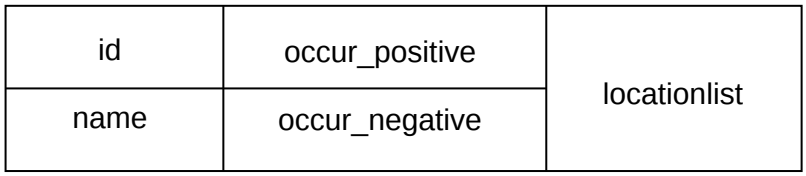


Figura 4.8: Registro prop_node

4.5 Lista de Localização

Note que o registro prop_node possui um campo do tipo location_list, uma lista que é utilizada para guardar todas as posições de uma proposição na árvore. Esse novo campo do registro é uma melhoria desenvolvida neste projeto. A implementação original do KSP não utilizava nenhuma estrutura como essa. Assim, para encontrar as ocorrências de uma determinada proposição, como na hora de verificar todas as posições de um literal que é puro na árvore para realizar simplificação (ver 4.7), era necessário percorrer toda a árvore em busca delas, um algoritmo com complexidade de ordem n, onde n é o número de símbolos lógicos, excetuando pontuação, que ocorre na fórmula de entrada. Com a nova implementação, no momento em que a proposição é criada na análise sintática, ou seja, sua estrutura tnode é gerada e inserida na árvore, essa posição é guardada dentro da Tabela de Símbolos no campo *locationlist de prop_node, o qual também é criado neste momento. Assim, para encontrar as posições de uma proposição no KSP modificado por este projeto, basta acessar o nó prop_node da proposição ao invés de percorrer toda a árvore da fórmula. Nós esperamos que essa melhoria seja significativa, pois as fórmulas com que o provador lida possuem centenas, milhares de nós na árvore, e evitar percorrimentos desnecessários deverá contribuir para o desempenho do programa. Em termos de complexidade, com o novo algoritmo economiza-se a busca pelas proposições na árvore, o que significa um ganho linear de ordem n.

É importante ressaltar que constantes lógicas que aparecem nas fórmulas de entrada também possuem lista de localização, elas são tratadas semelhantemente às proposições.

4.6 Cálculo da Polaridade

O KSP calcula a polaridade de cada proposição da fórmula de acordo com a Definição 7. A função que executa este cálculo é recursiva e realizada no pré-processamento. O pré-processamento é responsável por realizar o maior número de simplificações na fórmula, buscando um menor gasto computacional nas fases posteriores.

A função que calcula a polaridade possui 3 parâmetros: o tnode em questão, um inteiro *sign* indicando a polaridade (1 ou -1) e um inteiro representando o nível modal, não importante para os fins deste trabalho. A função busca encontrar uma proposição recursivamente. Se, no momento em que achar a proposição, o parâmetro *sign* for 1, o contador do prop_node *occur_posivite* é incrementado; caso contrário, o contador *occur_negative* é incrementado. No caso do *tnode* não ser uma proposição, a função é chamada recursivamente e o parâmetro *sign* é passado de acordo com a Definição 7 para cada tipo de subfórmula, positivo (1) ou negativo (-1). Após o cálculo da polaridade, é realizada a

eliminação de literal puro, pois de acordo com os valores dos contadores, será possível identificar se a proposição é um literal puro ou não.

4.7 Eliminação de Literal Puro e Simplificação

Na etapa de simplificação, faremos uso da lista de localização por nós implementada. Ao invés de percorrer toda a árvore em busca das proposições, basta fazer um laço de repetição com todas as proposições constantes desta lista. Isso é possível porque, na versão original do provador, a Tabela de Símbolos é implementada com o auxílio de uma variável global para um *hash*, o qual guarda todas as informações sobre as proposições. A biblioteca de macros para este *hash*, `uthash` [12], provê um iterador para percorrer todas as proposições.

Com esse laço, para cada proposição, testamos se esta proposição é um literal puro. Para isso, basta fazermos dois testes: primeiro, se o *occur_positive* é maior que zero e *occur_negative* é zero; ou se o *occur_negative* é maior que zero e *occur_positive* é zero. Se a proposição entrar no primeiro teste, ela é um literal puro positivo, então o nó correspondente na árvore sintática será substituído pela constante que representa verdade, ou seja, $\top$; se entrar no segundo teste, ela é um literal puro negativo, então o nó correspondente na árvore sintática será substituído pela constante que representa falso, ou seja, $\bot$. Se não entrar em nenhum teste, não é literal puro.

Caso seja um literal puro, o nó que era uma proposição será substituído por um nó do tipo constante. O id será falso ou verdadeiro, de acordo com a polaridade. Após feita essa primeira substituição, a simplificação será propagada de acordo com as regras presentes na Seção 3.1. A função que propaga a simplificação possui um laço de repetição que percorre a árvore desde a proposição enquanto houver mudança na árvore ou até o topo. Essa função é chamada passando o `tnode` que acabou de ser transformado em um constante, ou seja, um `tnode` com tipo constante.

Uma melhoria deste trabalho é o percorrimento da árvore também para cima, de um nó filho para um nó pai, pois na versão original do provador a única direção possível para se caminhar na árvore era para baixo. Então, dentro do laço de repetição, é testado o tipo do nó pai do `tnode` atual. Testando o tipo do nó pai, o próximo passo é verificar qual a constante filha deste nó. De acordo com a constante, se for possível a substituição de acordo com a Seção 3.1, é realizada a manipulação necessária para tal. Apresentamos, a seguir, um exemplo.

Seja $\phi = \Box_i p$ a fórmula de entrada. A estrutura de árvore da Figura 4.9 representa ϕ após a análise léxica e sintática, onde 100 é `id` de p.

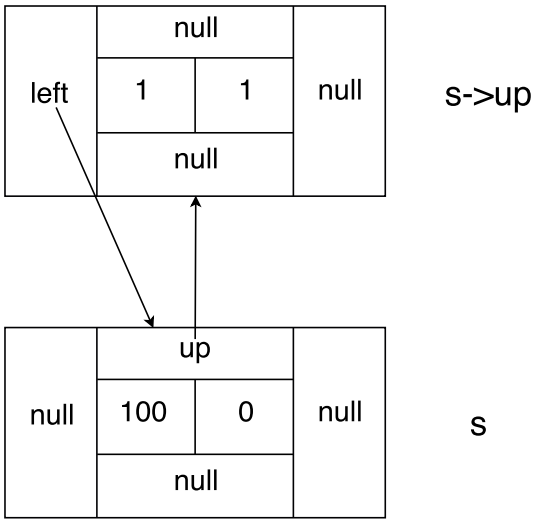


Figura 4.9: Representação da fórmula $\Box_i p$

Quando a proposição p é lida pelo nosso procedimento de eliminação de literal puro, ela é transformada em $\top$. A estrutura da árvore alterada é representada na Figura 4.10.

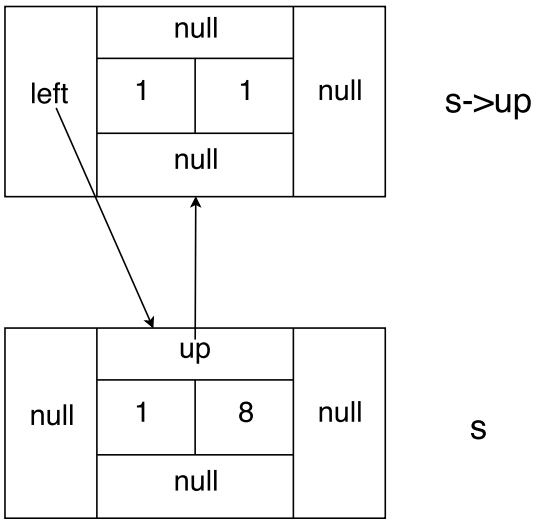


Figura 4.10: Representa da fórmula $\Box_i \top$

Após essa transformação, a propagação da constante, ou seja, a regra de simplificação correspondente, é aplicada. O tnode que representa $\top$ na fórmula seria o primeiro tnode analisado na propagação. O tipo de seu pai então é testado, no caso, $\Box_i$. A simplificação possível para uma fórmula do tipo $\Box_i$ é $\Box_i \top$, exatamente a fórmula que temos. Esta é simplificada para simplesmente $\top$. Com isso, a estrutura da árvore ficaria como na Figura 4.11.

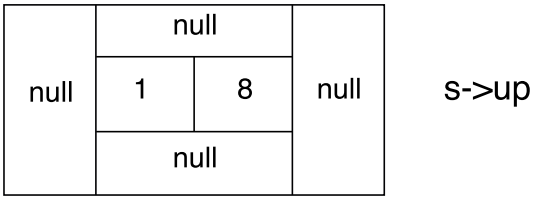


Figura 4.11: Representação da fórmula $\top$

Perceba que a árvore possuía dois tnode's e, ao final da simplificação, possui apenas um tnode. Isso é feito através de uma função que realiza a liberação de memória do tnode excluído da estrutura. Essa função, chamada free_tnode, além de liberar o espaço da memória corretamente, também exclui todas as ocorrências deste tnode na lista de localização, caso o tnode seja do tipo proposição ou uma constante. Isso evita que uma área da memória que tenha sido liberada pelo procedimento de simplificação seja acessada no laço de repetição que percorre as proposições, impedindo possíveis erros.

A execução da liberação de memória através de free_tnode é delicada, pois o iterador utilizado no laço de repetição da propagação é o mesmo tnode que será liberado, então se perdermos os dados do tnode na liberação, a propagação seria prejudicada. Para solucionar o problema, foi criada uma função que realiza a cópia de todos os atributos de tnode, copiar_tnode, evitando a perda de informação crítica para a execução da propagação. Essa função é recursiva e libera um nó da árvore e todos os nós abaixo deste, possuindo, portanto, complexidade de ordem n, onde n é o tamanho da fórmula.

O retorno da função de propagação indica se houve mudança na estrutura da árvore, pois, se houver, pode ser que outras proposições tornem-se puras. Por isso, se houver mudança, toda a rotina deve ser reexecutada, desde o cálculo da polaridade até a propagação das constantes. Se não houver mudança, tal repetição não se faz necessária.

Para cada simplificação de uma subfórmula, será mostrada a alteração sofrida nas estruturas de dados do programa nos próximos parágrafos.

Para o caso de $\Box_i\top$, descrito na Seção 3.1, haveria alteração nos nós `s`, nó filho, ($\top$) e `s->up`, nó pai, ($\Box_i$), da maneira descrita nas Figuras 4.9, 4.10 e 4.11. Ou seja, `s->up->type`, o tipo do nó pai, receberia o tipo da constante correspondente, isto é, 8, segundo a Tabela 4.1, e `s->up->id`, o identificador do nó pai, receberia 1, pois este é o valor definido para constante de valor *verdade* (Tabela 4.1). O tnode `s`, nó filho, deve ser excluído da árvore através da chamada à função free_tnode. Porém, antes de chamá-la, é necessário chamar a função para copiar_tnode, pelo motivo explicitado anteriormente.

Da mesma forma, para o caso de $\Diamond_i\bot$, os campos de `type` e `id` de `s->up`, sendo este o nó pai, ($\Diamond_i$) seriam alterados de mesmo modo, só que o `id` recebe o valor -1, pois representa a constante de valor $falso$. Neste caso, `s` ($\bot$) também seria excluído da árvore

com a função free_tnode.

No caso da negação, segundo as regras da Seção 3.1, para os dois casos, o `s` seria a constante em questão, $\bot$ ou $\top$, e o `s->up` seria a negação. O `s->up` se transforma na constante contrária a `s` e este `s` sai da árvore. `s->up->type` se transformaria em 8 e `s->up->id` receberia o contrário do `id` de `s`, isto é, se `s->id` for 1, `s->up->id` seria -1, e vice-versa.

A conjunção e a disjunção possuem um tratamento diferente, pois estas utilizam a estrutura formulalist ao invés de apenas nós filhos de esquerda e direita.

Se a constante $\top$ ocorrer em uma disjunção, esta pode ser reduzida para a constante $\top$. Para isso, o `s->up->type` (disjunção) é configurado como constante e seu `id` recebe o valor 1, indicando a constante $\top$. É necessário liberar toda a área de memória da lista, então a função free_formulalist é chamada e responsável por liberar o espaço de cada filho de formulalist. Porém, antes disso, há um percorrimento sequencial na lista da disjunção em busca de proposições, pois é necessário que as ocorrências de proposições dentro da lista tenham sua localização retirada da lista de localização que cada proposição possui, location_list. Para a conjunção é semelhante, a diferença é que ao invés de ser a constante $\top$, seria seu dual, $\bot$.

O outro caso possível para a disjunção é se a constante $\bot$ aparece na sua lista de nós filhos. Dessa forma, essa constante pode ser retirada da lista sem perda de informação. Isso é feito executando uma procura sequencial na lista de filhos em busca do tnode `s`; caso $\bot$ seja encontrado, uma *flag* é ativada para que a busca seja encerrada. No momento que a constante é encontrada, as ligações necessárias para o ajuste da lista são realizados e a área de memória correspondente ao nó é liberada. Após a exclusão da constante da lista de filhos de `s->up` (disjunção), é verificado se sua lista é vazia, pois, se for, `s->up` se tornará um nó que representa a constante $\bot$. A simplificação é realizada da mesma forma para conjunções, porém para o caso da constante $\top$ aparecer em sua lista de nós filhos.

Resta-nos realizar a simplificação da implicação e também da dupla implicação. A implicação é a subfórmula que nos permite maior variações de simplificações, quatro ao total. Em todos os casos, é realizada a liberação dos filhos da implicação com a função free_tnode, ou seja, o nó da esquerda, `s->up->left`, e o nó da direita, `s->up->right`.

O primeiro caso, onde `s->up->left->id` é igual a -1, ou seja, a constante $\bot$. Neste caso, `s->up->type`, anteriormente uma implicação, tornar-se-á a constante $\top$, alterando o campo `type` e `id` de `s->up`.

Já o segundo caso, onde o lado esquerdo é a constante $\top$, necessita de um tratamento maior. O tnode `s->up` irá receber as informações de `s->up->right`, então o `id` e o `type` deste serão copiados para aquele. Caso o `type` em questão seja uma proposição, uma nova ocorrência será adicionada a lista de localização desta proposição.

Se o lado direito da implicação for a constante $\top$, a simplificação é exatamente igual ao primeiro caso. Se o lado direito for a constante $\bot$, é semelhante ao segundo caso, porém `s` recebe as informações dos campos `type` e `id` de `s->up->left` e o `tnode s->up` torna-se uma negação, recebendo o `type` e `id` adequado.

Quando o lado direito de uma dupla-implicação é $\bot$, esta transforma-se na negação do lado esquerdo, ou seja, `s->up->type` e `s->up->id` recebem o valor de negação e `s->type` e `s->id` recebem estes campos de `s->up->left`. Caso o novo `s->id` seja referente a proposição, então essa nova aparição é adicionada a lista de localização da proposição.

A outra forma de simplificação com a dupla-implicação é quando seu filho da direita é $\top$. O `tnode s->up` irá copiar as informações de `s->up->left` e, se o `id` for uma proposição, sua ocorrência será adicionada à respectiva lista de localização.

Todas as mudanças das estruturas de dados do programa em cada etapa da simplificação estão resumidas na Tabela 4.2. Não foi criada uma tabela para resumir as simplificações da implicação e dupla-implicação pela complexidade dos testes e ações necessárias para simplificar essas fórmulas. Observe, entretanto, que estas modificações correspondem exatamente às regras de reescrita apresentadas na Seção 3.1.

Antes				Depois			
`s->type`	`s->id`	`s->up>type`	`s->up->id`	`s->type`	`s->id`	`s->up->type`	`s->up->id`
CONSTANTE	1	BOX	1	-	-	CONSTANTE	1
CONSTANTE	-1	DIAMOND	2	-	-	CONSTANTE	-1
CONSTANTE	1	NEGAÇÃO	3	-	-	CONSTANTE	-1
CONSTANTE	-1	NEGAÇÃO	3	-	-	CONSTANTE	1
CONSTANTE	1	CONJUNÇÃO	4	-	-	CONJUNÇÃO	4
CONSTANTE	-1	CONJUNÇÃO	4	-	-	CONSTANTE	-1
CONSTANTE	1	DISJUNÇÃO	5	-	-	CONSTANTE	1
CONSTANTE	-1	DISJUNÇÃO	5	-	-	DISJUNÇÃO	5

Tabela 4.2: Resumo das estruturas de dados na simplificação

Capítulo 5

Resultados

O provador foi executado sobre vários *benchmarks* para testar sua eficiência e verificar se houve a melhora esperada nos tempos de execução com a nova implementação da simplificação de eliminação de literal puro. Cada *benchmark* foi executado três vezes e a média dessas três medidas foram analisadas. Os testes foram executados em um computador com sistema operacional Ubuntu 14.04 (GNU/Linux 3.19.0-30-generic x86_64), processador Intel® Xeon® Processor E5-2620 v3, que possui frequência de *clock* 2.4GHz, 15M de memória *cache* e memória RAM de 64GiB.

5.1 Arquivos LWB iniciais

Os primeiros testes foram feitos em cima dos arquivos de *benchmarks* LWB [13]. Estes arquivos não são tão grandes, possuem entre 228 *bytes* e 125,3 *Kbytes*, e a maioria deles é executada em menos de um minuto. Foi estipulado um tempo limite para execução do programa de 62 segundos devido o tamanho dos arquivos. Ao todo, são 378 arquivos; destes, 189 contêm fórmulas satisfatíveis e 189, insatisfatíveis. Eles são divididos em 18 grupos/famílias com 21 arquivos cada. Ao final da execução deste *benchmark*, obtivemos os resultados apresentados na Tabela 5.1 que contém as médias dos tempos de execução de cada grupo de 21 arquivos para a nova implementação e o provador original.

O arquivo de configuração utilizado para a execução do provador com a nova implementação utilizou as seguintes opções de configuração: *populate_usable, max_lit_positive, snf++, bnfsimp, ordered, local, shortest, unit, lhs_unit, propdia, mres, mlple, forward, backward, limited_reuse_renaming, prenex, maxproof=1.* O arquivo de configuração do provador original possui as mesmas diretivas e adiciona a diretiva *early_ple*, para executar a eliminação de literal puro originalmente implementada.

Os resultados obtidos pela nova implementação foram todos corretos quanto à satisfatibilidade de cada fórmula. Os valores negativos da coluna “Diferença” da Tabela 5.1

Grupo de Arquivos	Original (s)	Projeto (s)	Diferença (%)
k_branch_n	23,4484	23,0400	1,77
k_branch_p	20,3962	20,1843	1,05
k_d4_n	0,0571	0,0551	3,75
k_d4_p	0,0124	0,0138	-10,34
k_dum_n	0,0070	0,0062	12,82
k_dum_p	0,0089	0,0076	16,67
k_grz_n	0,5083	0,5071	0,22
k_grz_p	0,0052	0,0040	32,00
k_lin_n	0,0079	0,0071	11,11
k_lin_p	0,0000	0,0000	0,00
k_path_n	0,0348	0,0524	-33,64
k_path_p	0,0302	0,0429	-29,63
k_ph_n	31,6230	31,5894	0,11
k_ph_p	31,8473	31,7668	0,25
k_poly_n	0,0221	0,0210	5,30
k_poly_p	0,0263	0,0256	3,11
k_t4p_n	0,0357	0,0357	0,00
k_t4p_p	0,0114	0,0119	-4,00

Tabela 5.1: Comparação lwb

indicam que a implementação deste projeto foi mais rápida do que o provador original, e valores positivos representam o contrário. Observamos que há uma variação grande no desempenho das implementações entre as diferentes famílias de fórmulas. Em alguns casos os tempos do provador original é melhor do que aquele com a nova implementação e em outros acontece o contrário. A maior diferença observada foi uma melhora de 33,64% nos arquivos `k_path_n`. A segunda maior diferença porém indica que o provador original obteve melhor desempenho nos arquivos `k_grz_p`. As diferenças são apresentadas em percentual porque os tempos de execução, para ambas as implementações, são muito baixos. Observamos, assim, que era melhor analisar mais dados para tirar conclusões a respeito de melhoria da eficiência do provador. O próximo passo foi realizar testes em arquivos maiores.

5.2 Arquivos LWB maiores

Com isso, buscamos novos arquivos de *benchmarks* que nos atendessem. Os novos arquivos utilizados foram gerados a partir do LWB [13] usando os parâmetros disponibilizados em [14], onde as fórmulas geradas são maiores do que a apresentação usual do LWB. Para este experimento, os arquivos gerados variam entre 627 *bytes* e 28,1 *Mbytes*. O *script* para geração dos *benchmarks* pode ser encontrado em [14]. A primeira tentativa foi gerar

os arquivos `grz` e `lin`, porém, quando executamos alguns arquivos nos provadores, o que levou alguns dias, notamos que a grande maioria não permitia simplificação por eliminação de literal puro, sendo assim uma amostra não boa para o fim deste projeto. A partir daí, geramos os arquivos do grupo *k_poly* e *k_t4p*, pois são mais indicados para nossos testes, já que permitem eliminação de literal puro. Estes novos arquivos tem até 28,1 *Mbytes*, enquanto que o *benchmark* passado tinha arquivos de no máximo 125,3 *Kbytes*, e requerem muito mais do desempenho do computador. A soma dos tamanhos dos 224 arquivos usados neste teste é de *1 Gbyte*. O tempo limite para a execução destes arquivos foi aumentado para mil segundos, pouco menos de 17 minutos. Ao total são 224 arquivos, 112 pertencentes à família *k_poly* e 112 pertencentes à família *k_t4p*. Os arquivos de configuração utilizados foram os mesmos do teste anterior.

Os resultados quanto à satisfatibilidade das fórmulas utilizadas neste estavam todos corretos, ou seja, a nova implementação apresentou a resposta devida para cada problema. Apresentamos a seguir os resultados discriminados por família de fórmulas. No que se segue, os tamanhos dos arquivos são dados em *bytes*.

5.2.1 k_poly

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_poly_n.0060	1,28E+05	1,44	1,38	4,34
k_poly_n.0080	2,21E+05	3,60	3,48	3,45
k_poly_n.0100	3,39E+05	7,85	7,10	10,51
k_poly_n.0120	4,83E+05	14,28	13,05	9,45
k_poly_n.0140	6,51E+05	23,95	21,32	12,37
k_poly_n.0160	8,45E+05	36,05	33,62	7,23
k_poly_n.0180	1,06E+06	52,74	48,25	9,31
k_poly_n.0200	1,31E+06	71,13	68,36	4,06
k_poly_n.0220	1,58E+06	97,47	97,86	-0,40
k_poly_n.0240	1,87E+06	130,46	131,16	-0,54
k_poly_n.0260	2,19E+06	164,38	167,04	-1,59
k_poly_n.0280	2,54E+06	224,19	218,39	2,66
k_poly_n.0300	2,91E+06	277,76	275,96	0,65
k_poly_n.0320	3,30E+06	344,91	340,07	1,42
k_poly_n.0340	3,72E+06	413,07	416,40	-0,80
k_poly_n.0360	4,17E+06	498,83	494,77	0,82
k_poly_n.0380	4,64E+06	590,00	599,57	-1,60
k_poly_n.0400	5,14E+06	761,93	669,31	13,84
k_poly_n.0420	5,66E+06	810,22	822,04	-1,44
k_poly_n.0440	6,21E+06	931,92	985,70	-5,46

Tabela 5.2: Comparação k_poly_n

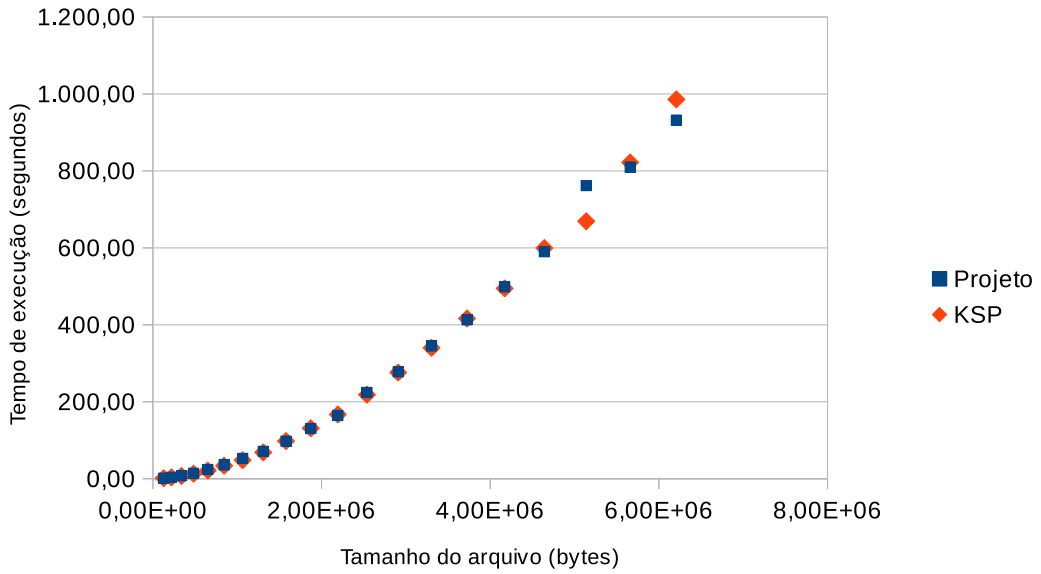


Figura 5.1: Gráfico da Tabela 5.2

Os resultados para as fórmulas da família `k_poly_n` são apresentados na Tabela 5.2, onde nota-se pouca diferença entre os tempos das duas implementações, com melhores resultados para o provador original. Para os arquivos menores, o provador original teve um melhor desempenho, obtendo até uma vantagem de tempo de 12,37% em relação à implementação deste projeto. As melhoras obtidas pela nova versão são pequenas, obtendo o maior ganho de tempo de 5,46%. O Gráfico 5.1 mostra esses valores visualmente e é possível notar que vários pontos se sobrepõem, o que indica que os valores de tempo destes pontos foram bem próximos para ambos os provadores. Vemos que os tempos se aproximam bastante e vão obtendo maiores diferenças para arquivos maiores. Os arquivos de `k_poly_n.460` até `k_poly_n.940` extrapolaram o tempo limite de execução e não foram levados em consideração. Vemos que para esse experimento, em 21 arquivos com tempos significativos, o provador original foi melhor para 13 destes, enquanto a nova implementação teve menor tempo em 7 arquivos. É importante notar que os arquivos que extrapolaram o tempo limite em uma versão também não conseguiram ser executados na outra.

Para as fórmulas da família *k_poly* que são não satisfatíveis, com resultados apresentados na Tabela 5.3, nota-se um equilíbrio entre os tempos, porém há quatro valores de diferença que se destacam: dois tempos com grande melhora, para os arquivos `k_poly_p.320` e `k_poly_p.340`, que alcançaram uma melhora de 44,64% e 40,03%, respectivamente; e dois tempos com a versão original sendo melhor que a nova, para os arquivos `k_poly_p.0080` e `k_poly_p.120`. O Gráfico 5.2 ilusta este equilíbrio entre os tempos e as grandes diferenças explicitadas anteriormente. As melhoras no tempo obti-

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_poly_p.0060	1,29E+05	1,57	1,57	0,00
k_poly_p.0080	2,23E+05	5,84	3,80	53,60
k_poly_p.0100	3,41E+05	7,80	7,67	1,65
k_poly_p.0120	4,85E+05	21,03	13,90	51,31
k_poly_p.0140	6,54E+05	22,37	22,62	-1,12
k_poly_p.0160	8,49E+05	34,59	34,40	0,54
k_poly_p.0180	1,07E+06	49,99	49,78	0,43
k_poly_p.0200	1,31E+06	72,65	70,93	2,43
k_poly_p.0220	1,58E+06	98,12	98,43	-0,31
k_poly_p.0240	1,88E+06	131,86	130,52	1,03
k_poly_p.0260	2,20E+06	112,86	114,29	-1,25
k_poly_p.0280	2,54E+06	145,62	143,56	1,44
k_poly_p.0300	2,91E+06	185,88	183,74	1,17
k_poly_p.0320	3,31E+06	230,75	416,82	-44,64
k_poly_p.0340	3,73E+06	281,38	469,16	-40,03
k_poly_p.0360	4,18E+06	338,48	333,08	1,62
k_poly_p.0380	4,65E+06	395,62	408,99	-3,27
k_poly_p.0400	5,15E+06	467,31	483,30	-3,31
k_poly_p.0420	5,67E+06	558,11	473,99	17,75
k_poly_p.0440	6,22E+06	657,66	649,21	1,30

Tabela 5.3: Comparação k_poly_p

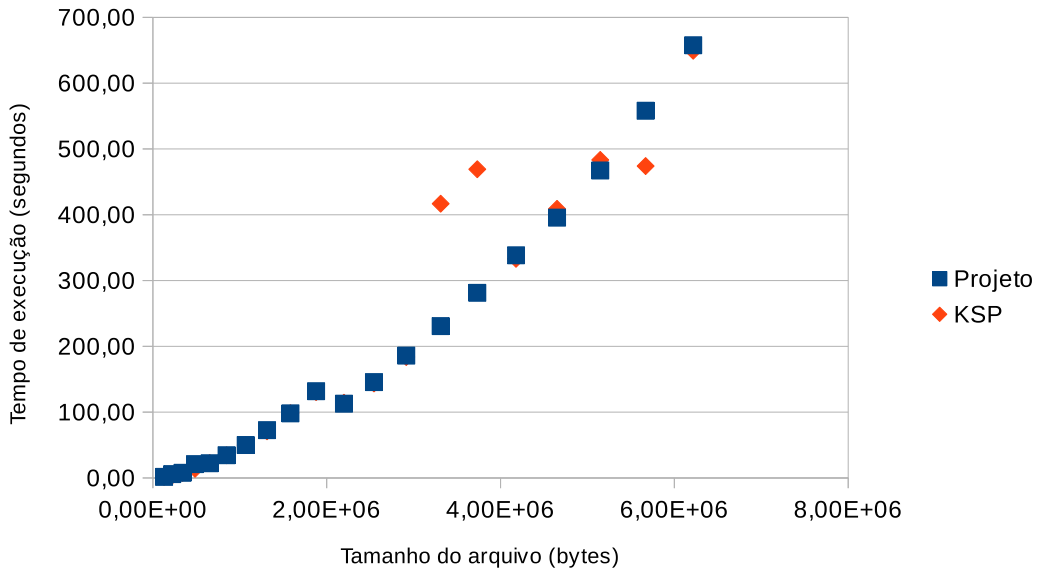


Figura 5.2: Gráfico da Tabela 5.3

das pela implementação deste projeto neste experimento deu-se em arquivos de tamanhos medianos para grandes, dando a ideia de que o provador na nova versão tende a conseguir melhores tempos para arquivos de maiores tamanhos.

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_t4p_n.0200	8,77E+04	2,930	2,933	-0,11
k_t4p_n.0300	1,32E+05	7,407	7,377	0,41
k_t4p_n.0400	1,75E+05	17,510	17,597	-0,49
k_t4p_n.0500	2,19E+05	38,063	44,910	-15,25
k_t4p_n.0600	2,63E+05	73,810	74,543	-0,98
k_t4p_n.0700	3,07E+05	149,677	124,450	20,27
k_t4p_n.0800	3,51E+05	182,723	193,837	-5,73
k_t4p_n.0900	3,94E+05	253,610	251,303	0,92
k_t4p_n.1000	4,38E+05	319,660	326,080	-1,97
k_t4p_n.1100	4,82E+05	395,883	398,893	-0,75
k_t4p_n.1200	5,26E+05	470,940	474,053	-0,66
k_t4p_n.1300	5,70E+05	551,230	551,690	-0,08
k_t4p_n.1400	6,13E+05	635,457	640,600	-0,80
k_t4p_n.1500	6,57E+05	725,063	735,663	-1,44
k_t4p_n.1600	7,01E+05	826,647	831,920	-0,63
k_t4p_n.1700	7,45E+05	945,843	952,483	-0,70

Tabela 5.4: Comparação k_t4p_n

5.2.2 k_t4p

O próximo grupo de arquivos contêm as fórmulas para a família *k_t4p*. Nota-se uma diferença significativa entre os tempos de execução para as fórmulas satisfatíveis (Tabela 5.4) e os tempos para as fórmulas não satisfatíveis (Tabela 5.5) dos dois provadores. Isso de dá pelo método de prova utilizado pelo provador, resolução, que, em geral, apresenta melhor desempenho para fórmulas não satisfatíveis [2].

A Tabela 5.4 mostra os valores de tempo de execução significativos para análise. Considerado o conjunto de 16 arquivos, a nova implementação teve melhor tempo em 13 deles. O melhor percentual de melhora foi de 15,25%. Já a piora foi de 20,27%. O Gráfico 5.3 ilustra estes valores e mostra a variação do tamanho dos arquivos. O tempo de execução ultrapassou o limite de 1000 segundos nos arquivos de `k_t4p_n.1800` a `k_t4p_n.9000`.

Para as fórmulas insatisfatíveis da família *k_t4p*, apresentadas na Tabela 5.5, a nova implementação obtém melhores tempos para os arquivos com tamanhos médio e grandes, da mesma forma que para a família *k_poly*, apresentada na subseção anterior. Os dados mostram 31 valores significativos para análise. Destes, 13 apresentam melhores tempos com a versão original e 18 com a nova versão. O Gráfico 5.4 mostra o crescimento do tamanho dos arquivos e que a diferença entre os tempos são pequenas, mas a maioria dos pontos que representam o melhor tempo são da nova implementação.

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_t4p_p.0300	6,61E+04	1,130	1,110	1,80
k_t4p_p.0400	8,80E+04	2,013	2,013	0,00
k_t4p_p.0500	1,10E+05	3,167	3,153	0,42
k_t4p_p.0600	1,32E+05	5,010	4,867	2,95
k_t4p_p.0700	1,54E+05	7,750	7,480	3,61
k_t4p_p.0800	1,76E+05	11,677	13,537	-13,74
k_t4p_p.0900	1,97E+05	17,287	17,183	0,60
k_t4p_p.1000	2,19E+05	24,650	23,920	3,05
k_t4p_p.1100	2,41E+05	34,000	34,210	-0,61
k_t4p_p.1200	2,63E+05	45,827	51,787	-11,51
k_t4p_p.1300	2,85E+05	60,027	60,037	-0,02
k_t4p_p.1400	3,07E+05	76,157	76,337	-0,24
k_t4p_p.1500	3,29E+05	100,713	95,247	5,74
k_t4p_p.1600	3,51E+05	122,963	113,137	8,69
k_t4p_p.1700	3,73E+05	131,370	132,343	-0,74
k_t4p_p.1800	3,95E+05	151,303	158,510	-4,55
k_t4p_p.1900	4,16E+05	174,287	175,330	-0,60
k_t4p_p.2000	4,38E+05	186,397	195,687	-4,75
k_t4p_p.2100	4,60E+05	203,467	216,207	-5,89
k_t4p_p.2200	4,82E+05	224,010	286,640	-21,85
k_t4p_p.2300	5,04E+05	248,120	261,780	-5,22
k_t4p_p.2400	5,26E+05	266,733	286,427	-6,88
k_t4p_p.2500	5,48E+05	294,017	307,733	-4,46
k_t4p_p.2600	5,70E+05	315,390	331,020	-4,72
k_t4p_p.2700	5,92E+05	341,953	359,690	-4,93
k_t4p_p.2800	6,14E+05	366,507	385,847	-5,01
k_t4p_p.2900	6,35E+05	390,733	416,613	-6,21
k_t4p_p.3000	6,57E+05	431,297	427,507	0,89
k_t4p_p.3500	7,67E+05	588,833	574,513	2,49
k_t4p_p.4000	8,76E+05	769,070	737,990	4,21
k_t4p_p.4500	9,86E+05	978,873	950,957	2,94

Tabela 5.5: Comparação k_t4p_p

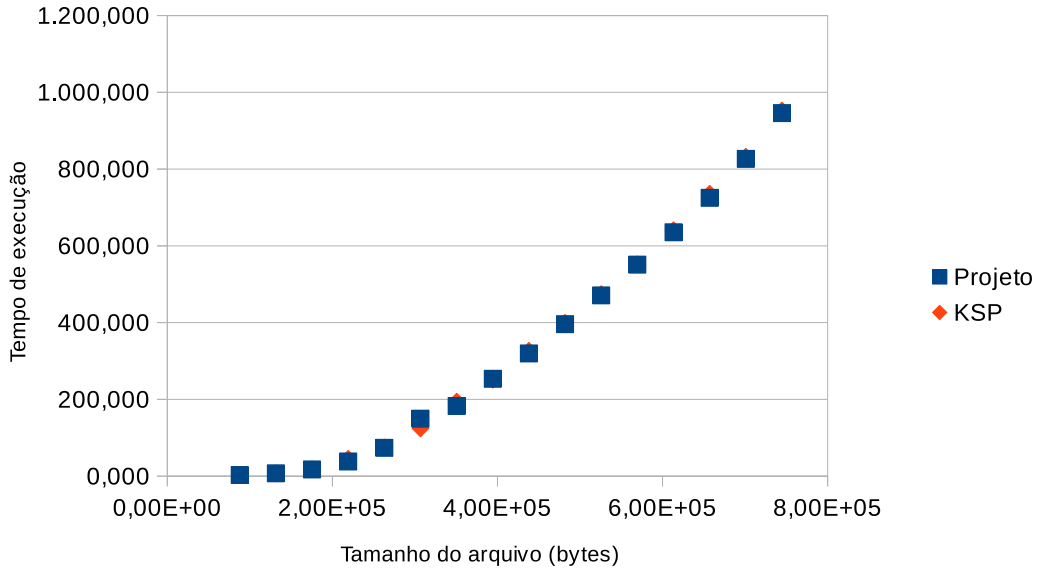


Figura 5.3: Gráfico da Tabela 5.4

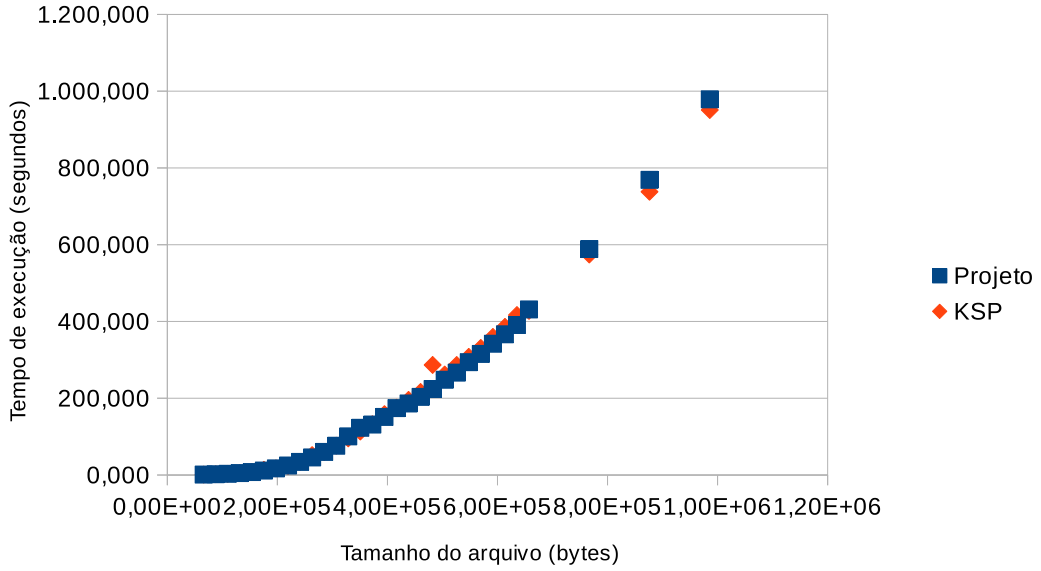


Figura 5.4: Gráfico da Tabela 5.5

5.3 Pré-processamento

As tabelas e resultados obtidos anteriormente foram com tempos de execução total do provador. Como os resultados não foram os esperados, fomos comparar apenas os tempos de pré-processamento, ou seja, apenas os tempos das simplificações realizadas, tanto a eliminação de literal puro como outras já implementadas no provador. Os arquivos utilizados para essa comparação foram aqueles que estouraram o tempo limite de

execução nos testes dos arquivos de teste para as famílias *k_poly* e *k_t4p*. O novo tempo limite foi de 480 segundos, estimativa de tempo necessária para execução desta parte do processamento, e os mesmos arquivos de configuração utilizados nos testes anteriores foram utilizados aqui. A única diferença foi a adição de uma nova diretiva aos arquivos de configuração que faz o provador executar apenas o pré-processamento (`ppi_only`).

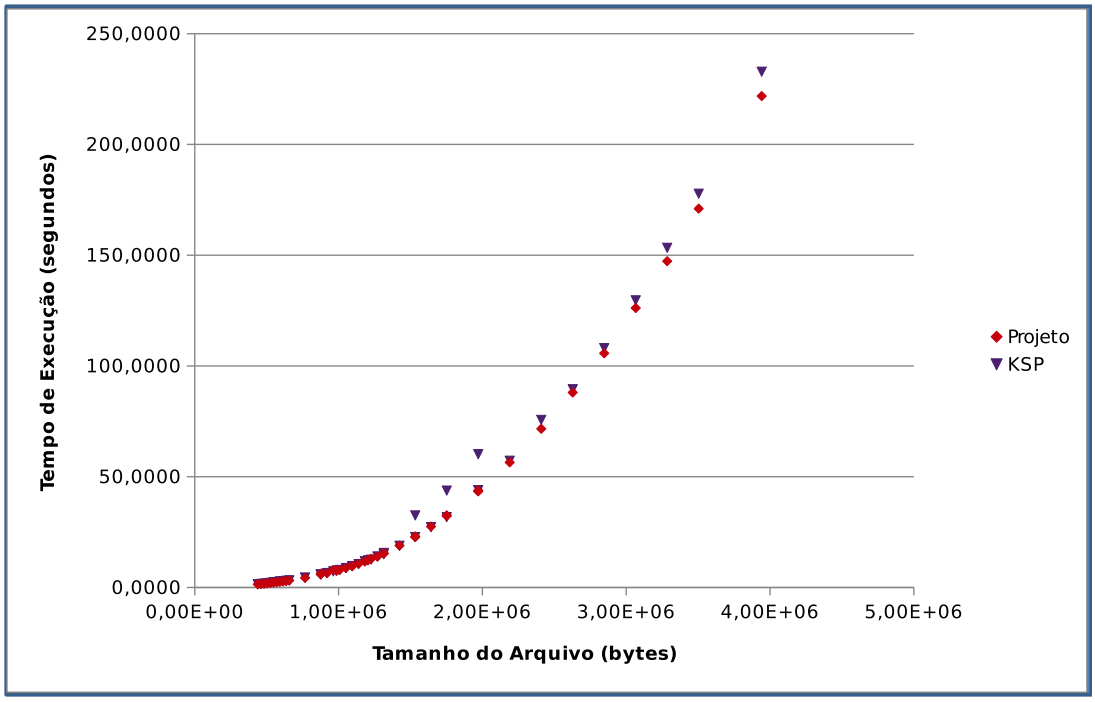


Figura 5.5: Gráfico da Tabela 5.6

O resultado deste experimento está na Tabela 5.6. Para todas as fórmulas constantes dos arquivos para a família *k_poly*, ambas as implementações ultrapassaram o tempo limite de execução. Já para as fórmulas da família *k_t4p*, a diferença entre os tempos obtidos pelas implementações são muito pequenas. A maioria dos resultados foram positivos, com a nova implementação obtendo melhora de tempo sobre o KSP. O Gráfico 5.5 representa os resultados desta amostragem.

5.4 Eliminação de Literal Puro

Como as diferenças de tempos obtidos nos experimentos anteriores foram sempre muito pequenas, decidimos testar apenas o tempo da realização de eliminação de literal puro. Nessa etapa não foi utilizado arquivo de configuração, pois não queríamos que o provador executasse nada além da simplificação de literal puro. Não houve tempo limite, pois possuíamos a hipótese de que a realização desta etapa fosse bem rápida. Os arquivos

testados foram os da família *k_poly* que não conseguiram executar no tempo limite do experimento de pré-processamento, descrito na seção anterior.

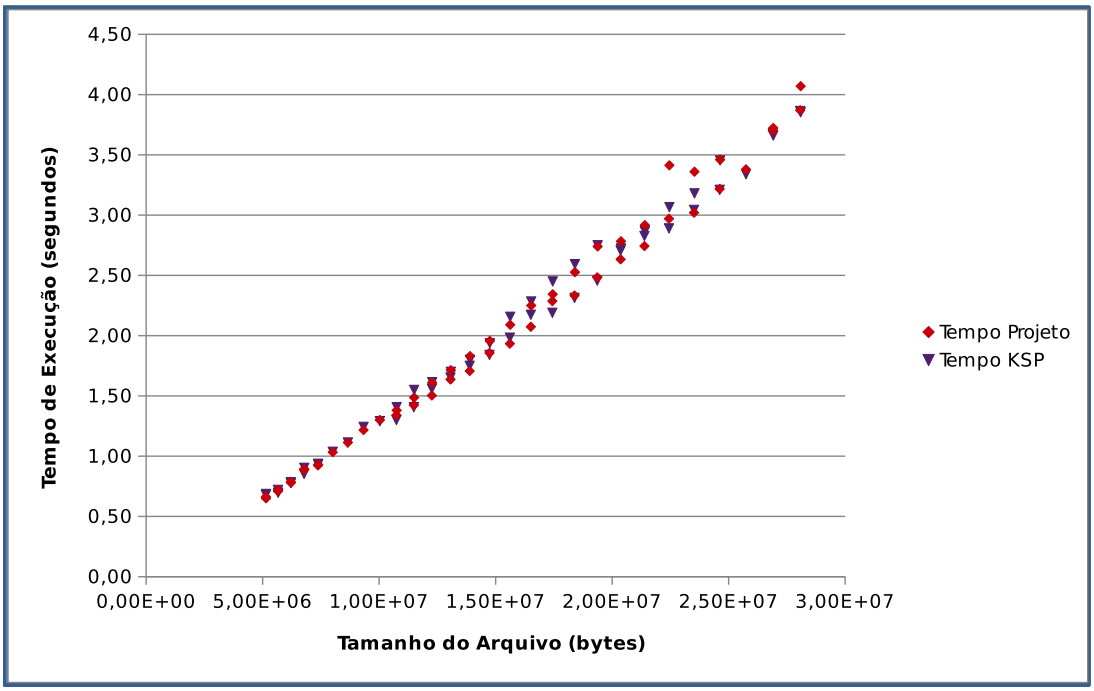


Figura 5.6: Gráfico da Tabela 5.7

A Tabela 5.7 mostra que a eliminação de literal puro é realizada rapidamente pelo provador. As maiores diferenças são na casa de 4% entre as versões, exceto um caso onde a nova versão obteve uma piora de 11,3%. Vemos que não há um padrão, mas para a maioria dos arquivos a nova implementação conseguiu executar seu procedimento mais rápido. Houve melhoria no tempo com a nova implementação em 24 dos 46 arquivos. Como não houve melhora em todos os arquivos, esse resultado pode indicar que para algumas fórmulas o algoritmo do provador original é melhor do que a nova implementação e vice-versa, pois de acordo com a fórmula, uma grande parte de nós de uma árvore podem ser eliminadas de uma só vez, como uma fórmula que possui uma disjunção de vários elementos e um destes é a constante lógica de verdade; desse modo, todo o nó seria reduzido à constante lógica $\top$.

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_t4p_n.2000	8,76E+05	5,9100	5,9167	-0,11
k_t4p_n.2100	9,20E+05	6,4767	6,5400	-0,97
k_t4p_n.2200	9,64E+05	7,8567	7,2933	7,72
k_t4p_n.2300	1,01E+06	7,8833	7,9067	-0,30
k_t4p_n.2400	1,05E+06	8,8200	8,7633	0,65
k_t4p_n.2500	1,10E+06	9,6633	9,6267	0,38
k_t4p_n.2600	1,14E+06	10,6633	10,5767	0,82
k_t4p_n.2700	1,18E+06	11,7300	11,8333	-0,87
k_t4p_n.2800	1,23E+06	12,9633	12,5833	3,02
k_t4p_n.2900	1,27E+06	13,9667	14,0167	-0,36
k_t4p_n.3000	1,31E+06	15,3133	15,0033	2,07
k_t4p_n.3500	1,53E+06	22,7567	22,7633	-0,03
k_t4p_n.4000	1,75E+06	32,2800	31,7467	1,68
k_t4p_n.4500	1,97E+06	43,3700	43,9400	-1,30
k_t4p_n.5000	2,19E+06	56,4800	57,2267	-1,30
k_t4p_n.5500	2,41E+06	71,6267	75,6533	-5,32
k_t4p_n.6000	2,63E+06	88,0200	89,4933	-1,65
k_t4p_n.6500	2,85E+06	105,7167	108,0300	-2,14
k_t4p_n.7000	3,07E+06	126,1600	129,6067	-2,66
k_t4p_n.7500	3,29E+06	147,2900	153,3433	-3,95
k_t4p_n.8000	3,50E+06	171,0233	177,7467	-3,78
k_t4p_n.9000	3,94E+06	221,8200	232,8400	-4,73
k_t4p_p.2000	4,38E+05	1,4367	1,4533	-1,15
k_t4p_p.2100	4,60E+05	1,5500	1,6167	-4,12
k_t4p_p.2200	4,82E+05	1,7000	1,7500	-2,86
k_t4p_p.2300	5,04E+05	1,8767	1,9333	-2,93
k_t4p_p.2400	5,26E+05	2,0200	2,0833	-3,04
k_t4p_p.2500	5,48E+05	2,2367	2,3733	-5,76
k_t4p_p.2600	5,70E+05	2,3933	2,4600	-2,71
k_t4p_p.2700	5,92E+05	2,5533	2,7500	-7,15
k_t4p_p.2800	6,14E+05	2,7700	2,8433	-2,58
k_t4p_p.2900	6,35E+05	2,9700	3,0300	-1,98
k_t4p_p.3000	6,57E+05	3,1567	3,2333	-2,37
k_t4p_p.3500	7,67E+05	4,3333	4,5100	-3,92
k_t4p_p.4000	8,76E+05	6,2900	5,8333	7,83
k_t4p_p.4500	9,86E+05	7,5600	7,5867	-0,35
k_t4p_p.5000	1,10E+06	9,7400	9,6933	0,48
k_t4p_p.5500	1,20E+06	12,3167	12,2433	0,60
k_t4p_p.6000	1,31E+06	15,4500	15,6000	-0,96
k_t4p_p.6500	1,42E+06	18,9300	18,8200	0,58
k_t4p_p.7000	1,53E+06	23,0500	32,5267	-29,14
k_t4p_p.7500	1,64E+06	27,5167	27,2667	0,92
k_t4p_p.8000	1,75E+06	32,5867	43,7467	-25,51
k_t4p_p.9000	1,97E+06	43,8267	60,1733	-27,17

Tabela 5.6: Comparação pré-processamento

Arquivo	Tamanho do Arquivo	Original (s)	Projeto (s)	Diferença (%)
k_poly_n.0400	5,14E+06	0,66	0,68	-2,45
k_poly_n.0420	5,66E+06	0,71	0,72	-1,39
k_poly_n.0440	6,21E+06	0,79	0,78	0,43
k_poly_n.0460	6,78E+06	0,89	0,85	3,91
k_poly_n.0480	7,38E+06	0,92	0,94	-1,42
k_poly_n.0560	1,07E+07	1,34	1,30	3,08
k_poly_n.0580	1,15E+07	1,42	1,41	1,18
k_poly_n.0600	1,23E+07	1,50	1,56	-3,63
k_poly_n.0620	1,31E+07	1,64	1,65	-1,01
k_poly_n.0640	1,39E+07	1,71	1,75	-2,48
k_poly_n.0660	1,47E+07	1,86	1,84	0,91
k_poly_n.0680	1,56E+07	1,93	1,98	-2,52
k_poly_n.0700	1,65E+07	2,07	2,17	-4,60
k_poly_n.0720	1,74E+07	2,29	2,19	4,41
k_poly_n.0740	1,84E+07	2,33	2,31	0,86
k_poly_n.0760	1,94E+07	2,48	2,46	1,09
k_poly_n.0780	2,04E+07	2,63	2,71	-2,71
k_poly_n.0800	2,14E+07	2,74	2,83	-2,95
k_poly_n.0820	2,24E+07	2,97	2,89	2,77
k_poly_n.0840	2,35E+07	3,02	3,04	-0,77
k_poly_n.0860	2,46E+07	3,22	3,21	0,21
k_poly_p.0400	5,15E+06	0,65	0,69	-5,34
k_poly_p.0420	5,67E+06	0,72	0,70	3,35
k_poly_p.0440	6,22E+06	0,78	0,78	0,43
k_poly_p.0460	6,79E+06	0,89	0,90	-1,11
k_poly_p.0480	7,39E+06	0,93	0,93	-0,36
k_poly_p.0500	8,01E+06	1,03	1,04	-0,32
k_poly_p.0520	8,66E+06	1,11	1,11	0,00
k_poly_p.0540	9,33E+06	1,22	1,24	-2,14
k_poly_p.0560	1,00E+07	1,30	1,29	0,78
k_poly_p.0580	1,08E+07	1,38	1,41	-1,90
k_poly_p.0600	1,15E+07	1,49	1,55	-4,09
k_poly_p.0620	1,23E+07	1,61	1,61	-0,21
k_poly_p.0640	1,31E+07	1,71	1,70	0,78
k_poly_p.0660	1,39E+07	1,83	1,80	1,67
k_poly_p.0680	1,48E+07	1,96	1,94	1,03
k_poly_p.0700	1,56E+07	2,09	2,16	-3,09
k_poly_p.0720	1,65E+07	2,25	2,28	-1,46
k_poly_p.0740	1,75E+07	2,34	2,45	-4,35
k_poly_p.0760	1,84E+07	2,53	2,59	-2,57
k_poly_p.0780	1,94E+07	2,74	2,75	-0,36
k_poly_p.0800	2,04E+07	2,78	2,72	2,20
k_poly_p.0820	2,14E+07	2,92	2,87	1,74
k_poly_p.0840	2,25E+07	3,41	3,07	11,30
k_poly_p.0860	2,35E+07	3,36	3,18	5,66

Tabela 5.7: Comparação eliminação de literal puro

Capítulo 6

Conclusão

A proposta deste trabalho foi implementar um algoritmo de eliminação de literal puro em fórmulas da Lógica Modal K e propagação de simplificação por meio de constantes lógicas em um provador de teoremas existente, buscando maior performance e menor tempo de execução. Para isso, a sintaxe e semântica da lógica clássica proposicional e da lógica modal K foram estudadas e revisadas, pois era necessário entender cada símbolo da linguagem para manipulá-los e saber de que forma poderiam ser simplificados. As simplificações possíveis foram retiradas de trabalhos que já as implementavam. A eliminação de literal puro também foi uma técnica estudada para que fosse possível aplicá-la da melhor forma no provador KSP. Este provador também foi um dos alvos de estudo: foi necessário conhecer os procedimentos já existentes e a forma que eles funcionavam, principalmente sua gramática, pois a inserção da lista de localização foi incluída em algumas regras da gramática, e essas regras só foram identificadas após este estudo e conhecimento do provador. A função free_tnode também já existia e foi modificada para adaptar-se à inclusão da lista de localização. Após todo o estudo, foi possível implementar o algoritmo estudado.

A implementação consumiu a maior parte do tempo dedicado à elaboração deste trabalho. Levou tempo para entender e dominar as estruturas de dados dinâmicas, como a árvore que guarda a fórmula de entrada, que já eram utilizadas por outras partes do programa. A quantidade de simplificações e a manipulação delas teve uma complexidade considerável, mas foi completada com sucesso, pois, em todos os testes realizados, a resposta quanto à satisfatibilidade das fórmulas tratadas foi correta, o que mostra que a simplificação está sendo realizada da forma como deve.

Com os dados levantados após os testes, é possível ver que a nova implementação não é sempre melhor em desempenho, pois em vários casos o provador original obteve tempos melhores do que os tempos da implementação deste projeto. Porém vemos uma real melhora de desempenho em vários dos testes com o algoritmo desenvolvido. Não houve

um arquivo que esta implementação conseguiu executar dentro dos limites de tempo estipulados que o provador original também não conseguiu executar, o que era desejável. Outra conclusão que tiramos é o fato de não haver um padrão predominante na melhora ou piora de tempo. Uma das hipóteses levantadas durante a análise dos resultados foi de que a nova implementação teria melhor desempenho em arquivos maiores e o provador original, em arquivos menores. De fato, isso ocorreu em algumas medições, mas as demais comparações realizadas mostraram que isso não era verdade. Com os resultados obtidos na análise do pré-processamento e da eliminação de literal, vimos que os resultados foram variados. Isso nos mostra que cada algoritmo de simplificação comporta-se melhor para uma estrutura de fórmula em particular. Uma grande diferença entre eles é que a nova implementação pode re-executar a rotina de eliminação de literais puros mais de uma vez, isso gera um custo maior no desempenho, porém o desempenho do provador original é compensado com mais simplificação na nova versão, o que contribui para que a processamento da fórmula nas próximas etapas aconteça mais rapidamente. É importante ressaltar que a eliminação de literal puro depende muito da fórmula de entrada. Existem fórmulas que vão permitir a eliminação de vários literais e outras que não irão permitir a eliminação de sequer um literal; com isso, fórmulas que permitem muita simplificação, tendem a realizar várias vezes o procedimento de eliminação de literais puros, pois sempre haverá mudança na árvore da fórmula.

Os resultados obtidos mostraram que a nova implementação consegue um melhor tempo em alguns casos, porém essa melhora não é substancial. O algoritmo mostrou um bom comportamento: embora os tempos sejam próximos dos do provador original, em alguns casos a nova implementação foi melhor. O algoritmo implementado respeita tempos polinomiais de execução, um dos objetivos deste trabalho. O fato de realizar mais simplificação e manter os tempos bem próximos, às vezes melhor, que o provador original corrobora para concluirmos que este trabalho alcançou seus objetivos. Era esperado que o tempo de execução fosse ainda menor, devido à implementação da lista de localização, mas experimentalmente a tese não foi confirmada para todos os casos. Próximos trabalhos podem ser realizados executando este algoritmo para arquivos ainda maiores dos que foram testados neste projeto, da grandeza de gigas; arquivos de outros grupos de *benchmarks* e melhorando pontos como a busca de um nó dentro da lista de uma conjunção ou disjunção.

Referências

[1] Chellas, Brian F.: *Modal Logic: an introduction*. Cambridge University Press, New York, 1980. 1, 5, 7

[2] Nalon, Cláudia, Ullrich Hustadt e Clare Dixon: *Ksp: A resolution-based prover for multimodal K*. Em Olivetti, Nicola e Ashish Tiwari (editores): *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 de *Lecture Notes in Computer Science*, páginas 406–415. Springer, 2016, ISBN 978-3-319-40228-4. `http://dx.doi.org/10.1007/978-3-319-40229-1`. 1, 9, 13, 14, 31

[3] Ladner, Richard E.: *The computational complexity of provability in systems of modal propositional logic*. SIAM J. Comput., 6(3):467–480, 1977. 1

[4] SIPSER, M.: *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996. 1

[5] Nonnengart, Andreas e Christoph Weidenbach: *Computing small clause normal forms*. Em Robinson, Alan e Andrei Voronkov (editores): *Handbook of Automated Reasoning*, volume I, capítulo 6, páginas 335–367. Elsevier Science B.V., 2001. 1, 10

[6] Nalon, Cláudia e Clare Dixon: *Anti-prenexing and prenexing for modal logics*. Em Fisher, Michael, Wiebe van der Hoek, Boris Konev e Alexei Lisitsa (editores): *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 de *Lecture Notes in Computer Science*, páginas 333–345. Springer, 2006, ISBN 3-540-39625-X. `http://dx.doi.org/10.1007/11853886_28`. 1, 5, 6, 9

[7] Huth, Michael e Mark Ryan: *Lógica em Ciência da Computação: Modelagem e Argumentação dos Sistemas*. LTC, Rio de Janeiro, 2005. 3, 4, 5

[8] Raman, Vasumathi, Cameron Finucane e Hadas Kress-Gazit: *Temporal logic robot mission planning for slow and fast actions*. Em *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, páginas 251–256. IEEE, 2012. 3

[9] Nalon, Cláudia: *Notas de aula: Lógica Computacional 1*. Manuscrito, 2015. 5, 6, 9, 11

[10] Davis, Martin e Hilary Putnam: *A computing procedure for quantification theory*. Automation of Reasoning, página 125–139, 1983. 11

[11] Szwarcfiter, J. L. e Markenzon, L.: *Estruturas de Dados e Seus Algoritmos*. Em *I Escola Brasileira de Otimização*, Rio de Janeiro, 1989. 13, 14, 16

[12] Hanson, Troy D.: *uthash: a hash table for C structures*, 2016. Disponível em `http://troydhanson.github.io/uthash/index.html`. 21

[13] Jaeger, G., P. Balsiger, A. Heuerding, S. Schwendimann, M. Bianchi, K. Guggisberg, G. Janssen, W. Heinle, F. Achermann, A. D. Boroumand, P. Brambilla, I. Bucher e H. Zimmermann: *LWB–The Logics Workbench 1.1*. http://www.lwb.unibe.ch/, 2016. University of Berne, Switzerland. 26, 27

[14] *Script para geração de arquivo de benchmark*. Disponível em `http://www.cic.unb.br/~nalon/software/generate_lwb_files`, 2016. Acessado em: 2016-12-04. 27